[redacted]

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GCS/ENG/90D-06

ACCESS AND OPERATOR METHODS FOR THE TRITON NESTED RELATIONAL DATABASE SYSTEM

THESIS

Tina Marie Harvey
Captain, USAF

AFIT/GCS/ENG/90D-06

AFIT/GCS/ENG/90D-06

ACCESS AND OPERATOR METHODS FOR THE TRITON NESTED RELATIONAL DATABASE SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)

Tina Marie Harvey, B.S.

Captain, USAF

December 13, 1990

Acknowledgments

Before I recognize the people who significantly contributed to this research effort, I must first give praise to God for His comforting love and righteousness. I would like to thank my instructors at AFIT for providing me a solid background for this research. For guiding and reviewing my work, I would like to acknowledge the members of my committee, Captain Eric Hanson and Dr. Gary Lamont. I am especially indebted to my advisor, Major Mark Roth, who encouraged me and directed this research effort. His superb guidance and invaluable knowledge in the realm of database research and nested relational theory played a critical role in the success of this thesis. I would like to thank Major Martin Stytz, my reader, for never failing to help and encourage me; his enthusiasm is only equalled by his love for red pens. For providing further information and help, I would like to recognize Captain Craig Schnepf, my partner on the Triton project. I would also like to thank my roommate, Captain Nagin Ahmed, for always being there for me these last eighteen months. Of course, I cannot say thank you enough to my husband, Michael, whose love and selflessness has taught me the greatest lesson of all.

Tina Marie Harvey

Table of Contents

Page

Acknowledgments ii

Table of Contents iii

List of Figures viii

List of Tables x

Abstract xi

I. Introduction 1-1

1.1 Overview 1-1

1.2 Background 1-2

1.2.1 Relational Database Model. 1-2

1.2.2 The Nested Relational Model. 1-2

1.3 Purpose of Thesis 1-4

1.3.1 The Triton Nested Relational Database System Architecture. 1-5

1.3.2 Algebra Used in the Triton System. 1-6

1.4 Scope of this Research/Objective 1-6

1.5 Methodology/Approach 1-8

1.6 Materials and Equipment 1-8

1.7 Outline of this Document 1-9

II. Overview of the Nested Relational Model and the EXODUS Extensible Database System 2-1

2.1 Overview 2-1

2.2 Impetus for the Development of the Nested Relational Model 2-1

Page

2.3 Historical Development of the Nested Relational Model . . . 2-2

2.3.1 The Relational Model. 2-2

2.3.2 Introduction of Set-Valued Attributes. 2-2

2.3.3 Introduction of Relation-Valued Attributes. 2-3

2.4 Notable Advances in Nested Relational Database Theory . . 2-6

2.4.1 The SQL/NF Query Language. 2-6

2.4.2 Colby Algebra and its Optimization. 2-7

2.4.3 Indexing Techniques and Access Methods. 2-7

2.5 Related Work 2-9

2.5.1 Advanced Information Management Prototype . . . 2-9

2.5.2 Darmstadt Database System 2-9

2.5.3 The Verso DBMS. 2-10

2.6 EXODUS Extensible Database System 2-10

2.6.1 EXODUS Tools Used in the Production of the Triton Nested Database System. 2-11

2.7 Summary 2-13

III. Design and Implementation 3-1

3.1 Overview 3-1

3.2 Representation of Nested Relations in Triton 3-2

3.2.1 E Representation of Non-Nested Relations. 3-2

3.2.2 E Representation of Nested Relations. 3-2

3.3 Design and Function of Methods 3-5

3.3.1 The Filescan Method. 3-5

3.3.2 The Loops_Join Method. 3-8

3.3.3 Store_Values Method. 3-12

3.3.4 Modify Method. 3-13

3.3.5 Delete Method. 3-15

Page

3.4 Code Generator 3-16

3.4.1 Rationale Behind Design Decisions for the Code Generator. 3-16

3.4.2 System Catalogs. 3-16

3.4.3 Operation of the Code Generator. 3-17

3.5 Summary 3-19

IV. Analysis and Evaluation 4-1

4.1 Overview 4-1

4.2 Comparison of the NRM and Relational Representations of the IDEF$_0$ Language Data 4-1

4.2.1 Schema Definitions. 4-1

4.2.2 Queries Used in this Comparison. 4-2

4.2.3 Method of Comparison. 4-2

4.2.4 Comparison of Code Generation Times. 4-3

4.2.5 Comparison of Query Execution Times. 4-5

4.2.6 Summary of Comparison. 4-8

4.3 Evaluation of the EXODUS Toolkit in this Research Effort . 4-9

4.3.1 Collections. 4-9

4.3.2 Classes. 4-10

4.3.3 Persistence. 4-10

4.3.4 Data Representation. 4-10

4.3.5 Optimizer Generator. 4-10

4.3.6 Storage Manager. 4-11

4.3.7 Overall Evaluation of the EXODUS Toolkit. 4-11

4.4 Summary 4-12

Page

V. Conclusions and Recommendations 5-1

5.1 Overview 5-1

5.2 Summary of Research 5-1

5.3 Conclusions 5-2

5.4 Recommendations 5-3

Appendix A. Colby Relational Algebra A-1

A.1 Select (σ). A-1

A.2 Project (π). A-2

A.3 Join ($\bowtie$). A-3

A.4 Nest (ν). A-4

A.5 Unnest (μ). A-5

Appendix B. Definition of Data Structures B-1

B.1 Plan Node B-1

B.2 Pred Node B-2

B.3 List Node B-3

B.4 Attrdesc Node B-5

Appendix C. Data for the Employees Relation C-1

Appendix D. Data for the Products Relation D-1

Appendix E. Filescan Method E-1

Appendix F. Loops_Join Method F-1

Appendix G. Store_Values Method G-1

Appendix H. Modify Method H-1

Appendix I. Delete Method I-1

Page

Appendix J. SQL/NF Create Table Definition for the Relational Version of the IDEF$_0$ Language Data J-1

Appendix K. SQL/NF Create Table Definition for the Nested Version of the IDEF$_0$ Language Data K-1

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure	Page
1.1. The Flat *Employee* Relation Without Children	1-3
1.2. The *Children* Relation	1-4
1.3. The Nested *Employee* Relation	1-4
1.4. The Triton Nested Database System Architecture	1-5
2.1. Representation of a Relational VHDL Database	2-3
2.2. Representation of a VHDL Database with Set-Valued Attributes	2-4
2.3. Representation of a VHDL Database with Relation-Valued Attributes	2-5
2.4. The Triton Nested Database System Architecture Using EXODUS	2-12
3.1. Prototype Nested Database Backend	3-1
3.2. The *Children* Relation	3-3
3.3. E Specification of the *Children* Relation	3-3
3.4. The Nested *Employee* Relation	3-4
3.5. E Specification of the *Employee* Relation	3-4
3.6. Example Plan Tree for *Filescan* Query	3-7
3.7. Example Plan Tree for *Loops_Join* Query	3-10
3.8. Example Plan Tree For *Store_Values* Query	3-13
3.9. Example Plan Tree For *Modify* Query	3-14
3.10. Example Plan Tree For *Delete* Query	3-15
3.11. Contents of *Rel_table*	3-17
3.12. Contents of *Sym_table*	3-18
A.1. Employees Over 35 With Male Children	A-2
A.2. Project Out Employee and Child. ən Names	A-2
A.3. The *Student* Relation	A-3

Figure Page

A.4. The *Employee* Relation Joined (on Children) to the *Student* Relation . A-4

A.5. The Flat *Employee* Relation A-4

A.6. The Nested *Employee* Relation A-5

B.1. A *Plan* Node B-2

B.2. A *Pred* Node B-2

B.3. A Predicate Tree B-4

B.4. A *List* Node B-4

B.5. An *Attrdesc* Node B-5

C.1. Schema for the *Employees* Nested Relation C-1

D.1. Schema for the *Products* Nested Relation D-1

List of Tables

Table		Page
4.1.	Code Generation Times for Relational Drawing Data Query	4-3
4.2.	Code Generation Times for Relational Activity Data Dictionary Query	4-3
4.3.	Code Generation Times for Relational Data Element Data Dictionary Query	4-4
4.4.	Code Generation Times for Nested Queries	4-4
4.5.	Comparison of Code Generation Times	4-5
4.6.	Execution Times for Relational Drawing Data Query	4-6
4.7.	Execution Times for Relational Activity Data Dictionary Query	4-6
4.8.	Execution Times for Relational Data Element Data Dictionary Query .	4-7
4.9.	Query Execution Times for Nested Queries	4-7
4.10.	Comparison of Query Execution Times	4-8

AFIT/GCS/ENG/90D-06

Abstract

Unique database requirements in the realm of non-standard applications (such as computer-aided design (CAD), computer-aided software engineering (CASE), and office information systems (OIS)) have driven the development of new data models and database systems based on these new models. In particular, the goal of these new database systems is to exploit the advantages of complex data models that are more efficient (in terms of time and space) than their relational counterparts.

This research effort describes the design and implementation of the Triton nested relational database system, a prototype system based on the nested relational data model. Triton is intended to be used as the backed storage component for some non-standard application. To quickly prototype the system, the EXODUS extensible database system is used in the development of Triton.

The research presented in this document focuses on Triton's operator and access methods, and compares the performance of the nested relational model versus the relational model using these methods. In addition, the effectiveness of the EXODUS extensible database toolkit is evaluated.

ACCESS AND OPERATOR METHODS FOR THE TRITON NESTED RELATIONAL DATABASE SYSTEM

I. Introduction

1.1 Overview

Information storage and retrieval plays an increasingly important role in our lives. Banking transactions, military command and control, and hospital record keeping are but a few examples of our dependence upon the prompt processing and efficient management of large volumes of data.

Since the early 1960's, computerized databases have proven to be an indispensible way to manage information (19). A database management system (DBMS) consists of two parts: a set of computer data and programs that provide access to that data (18:1). A DBMS is different from a file system because in a file system only small portions of data can be examined or updated at a time; file systems do not allow the efficient correlation of data in separate files. DBMSs provide this correlation in a fast and easy fashion.

In recent years, database research has focused on the development of database systems to support non-standard applications, such as computer-aided design (CAD), computer-aided software engineering (CASE), and office information systems (OIS). Requirements in these new application areas have driven the development of new data models and database systems based on these new models to efficiently manage large volumes of non-standard, or complex, data (such as textural or pictoral information).

The Triton nested relational database system is a prototype system based on an extension of the relational model (8), called the *nested relational model* (32), which allows a hierarchical representation of complex objects. The primary goal of this research effort is to develop and implement operator and access methods for the Triton system that efficiently exploit this hierarchical representation of data. The intended use of the Triton system is as the backend for applications which require database support not provided by current relational databases, such as CASE or CAD tools. Thus, queries to the system will be made via embedded calls within the application program, and, as such, Triton will be dealing with known queries as opposed to *ad hoc* queries.

The development of a DBMS is a mamoth task in terms of time and the amount of programming required. In order to quickly prototype the Triton system we utilized EXODUS (7), an extensible database "toolkit" developed at the University of Wisconsin. EXODUS provides several necessary facilities and tools that aid in the development of Triton's access methods, which include a storage manager and a persistent programming language, called E. Not only does this research effort provide Triton the ability to process queries of any type (except for data restructuring queries), it also provides the means to evaluate the advantages and disadvantages of using extensible systems such as EXODUS and a persistent programming language to build application-specific database management systems.

The purpose of this chapter is to define the scope and purpose of this research effort. The relational model is summarized, followed by an introduction of the nested relational model. The Triton nested database system is then described, which leads into the purpose and scope of my research.

1.2 Background

1.2.1 Relational Database Model. In 1970, Codd proposed the formal relational database model (8). In this model, information is stored in a tabular format called a *relation* which is composed of attributes, or fields, and records, or tuples. A column in a table represents an attribute and a row is a single record. As an example, Figure 1.1 shows a simple relation depicting information on employees of a company. In the relational model, all attributes must be atomic, or undecomposable. In other words, an attribute can only contain one integer, one real number, or a single character string. "A relation which only has atomic valued attributes is said to be in first normal form (1NF)." (23:7)

1.2.2 The Nested Relational Model. The nested relational model (NRM) is an extension of the relational model that relaxes the 1NF restriction and allows attributes of a

dept	*emp_name*	*emp_age*	*emp_ssn*
Marketing	John Smith	27	237-46-3567
Research	Michael Taylor	31	395-73-8901
Advertising	Tina Therrien	43	555-12-3434
Personnel	Carla Dunlap	37	624-35-8152

Figure 1.1. The Flat *Employee* Relation Without Children

relation to be a set of values (20), or possibly, another relation (32). The NRM has two advantages over the relational model for the storage of hierarchical data: decreased storage requirement and increased processing speed. These advantages are best illustrated by an example. Assume we wish to store information on the employees of a company and their children. If a relational database is used to store this information, it requires the relational tables shown in Figures 1.1 and 1.2. Notice that the *emp_ssn* attribute is duplicated in both tables so that the correspondence between employees and their children is not lost. This attribute duplication clearly requires additional data storage. However, if a nested relational database is used to store this information, only a single table is required as depicted in Figure 1.3. Because children are nested directly under employees in the same table, the *emp_ssn* field does not have to be duplicated to correlate employee data and children data. In addition, a query on the relational database involving both employee data and children data would require the two tables to be joined, which takes time.[1] Since all data is kept in one table in the nested relational model, a costly join does not have to take place. An intangible advantage of the NRM over the relational model is a more intuitive mapping of complex data; the relational model splits the data into 1NF "chunks", while the NRM allows data to remain in hierarchical form within the database schema.

However, the nested relational model does have some disadvantages. The schema of relations is more complex due to relation-valued attributes. In addition, because data is not normalized, updates to a nested relation may require the modification of several tuples as opposed to the modification of one tuple in the relational version. Finally, the nested relational model does not allow the sharing of relation-valued attributes between

[1]Such a query might be, "What are the names of Michael Taylor's children?"

emp_ssn	*child_name*	*child_age*	*sex*
237-46-3567	Jeramie	8	M
237-46-3567	Todd	4	M
395-73-8901	Susan	3	F
555-12-3434	Laura	18	F
555-12-3434	John	13	M
555-12-3434	Matthew	11	M
624-35-8152	George	5	M
624-35-8152	Janis	3	F

Figure 1.2. The *Children* Relation

dept	*emp_name*	*emp_age*	*emp_ssn*	*children*		
				child_name	*child_age*	*sex*
Mktg	J. Smith	27	237-46-3567	Jeramie	8	M
				Todd	4	M
Rsrch	M. Taylor	31	395-73-8901	Susan	3	F
Adv	T. Therrien	43	555-12-3434	Laura	18	F
				John	13	M
				Matthew	11	M
Pers	C. Dunlap	37	624-35-8152	George	5	M
				Janis	3	F

Figure 1.3. The Nested *Employee* Relation

tuples of nested relations. However, for the Triton system, we feel the advantages of the nested relational model outweigh its disadvantages, particularly since the nested model outperforms the relational model for complex data in terms of query execution time (as shown in Chapter 4).

1.3 Purpose of Thesis

The goal of this research is to provide Triton the ability to process queries of any type (except for data restructuring queries) and to lay the foundation for future work in this area. In particular, this research centers on the design and implementation of access and operator methods that take advantage of the hierarchical nature of the nested data. Because the EXODUS extensible database system is used in the development of the Triton

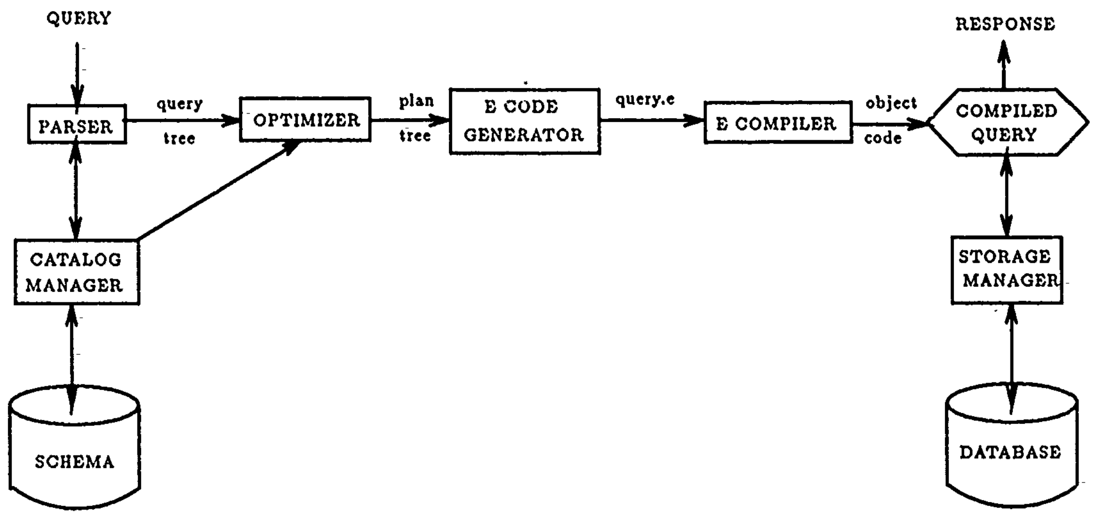

Figure 1.4. The Triton Nested Database System Architecture

system, the operator and access methods are developed within the constraints dictated by EXODUS. The two areas where these constraints are felt most strongly are in the representation of the nested relational model using the persistent programming language constructs of the E programming language provided by EXODUS (25), and access to these persistent structures via the EXODUS storage manager (5). These issues are discussed in more detail in Chapter 4.

1.3.1 The Triton Nested Relational Database System Architecture. The scope of this research effort is best explained within the context of the operation of the Triton system. Query processing in this system, depicted in Figure 1.4, is explained below.

The user (or application program) inputs a query that is written in a special query language for nested relations, called SQL/NF (26). By accessing the database schema via the catalog manager, the parser translates the query into an unambiguous algebra for computer manipulation. This algebraic representation is contained in a data structure called a *query tree*, which consists of a series of nodes and pointers to other nodes. Each query node contains a relational algebra operator or an access method, the relation(s) the method refers to, and any Boolean conditions to apply.

The query tree is passed to the rule-based optimizer where it is changed to a plan tree. In the optimizer, the sequence and contents of the nodes in the query tree are changed into an optimized plan tree and relational operators are replaced with specific operator methods.

The E code generator traverses the plan tree and generates the E code to effect the operator and access methods to perform the query. The E compiler links the methods with the E code that specifies the relation definitions. The resulting object code is executed to manipulate the information in the database via the storage manager. The manipulated data is returned to the user as the response to the query.

1.3.2 Algebra Used in the Triton System. The implementation of the operator and access methods relies on a structural representation of the nested relational algebra developed by Latha Colby (9). Colby defined a recursive algebra for nested relations that builds on the traditional relational algebra operators. Her operators support the retrieval of information from any level of nesting in a relation without first "flattening out" the relation to fit the relational model (9:276).

Colby's algebra uses an extensive group of set operators and redefines the *select* (σ), *project* (π), and *join* ($\bowtie$) operators of the relational model for use on nested objects. In addition, she defined two new operators, *nest* (ν) and *unnest* (μ). The nest operator restructures the database into a nested form, while the unnest operator "flattens out" the nested relation into a relational form. Appendix A gives a brief summary of each of Colby's relational operators and demonstrates their use on example nested relations.

1.4 Scope of this Research/Objective

The work accomplished in this research effort is summarized as follows:

1. Design and implementation of the operator methods of the Colby algebra to handle multiple levels of nesting, including:
    - The *filescan* method that implements a project and/or a select
    - The *loops_join* method that implements a join as well as a project and/or select

2. Design and implementation of the access methods to modify data in the database, including:

   - The *store_values* method that adds data to a relation at any level or levels of nesting
   - The *modify* method that modifies data in a relation at any level or levels of nesting
   - The *delete* method that deletes data from a relation at any level or levels of nesting
   - The *create_rel* method that adds a new relation to the database
   - The *drop_rel* method that deletes a relation from the database

3. Design and implementation of the code generator, *codegen*, to implement the operator and access methods

4. Testing of methods and code generator

5. Comparison of the performance of the nested relational model versus its normalized (1NF) version using the operator methods and code generator:

   - Development of an E representation of a sample set of relational IDEF_0 language data (23)
   - Development of an E representation of nested relational IDEF_0 language data (23)
   - Design of programs to load data into both E representations
   - Creation of queries for both representations to evaluate performance of the NRM against the traditional relational model
   - Comparison of the NRM against the relational model based on code generation time and query execution time

6. Discussion of the advantages and disadvantages of using the EXODUS toolkit in this research effort

1.5 Methodology/Approach

EXODUS provides several facilities and tools for the construction of application-specific DBMSs. One such tool is the E programming language and its compiler (25, 24) which is an extension of C++ (31) and builds on the object-oriented nature of C++. By exploiting the powerful object-oriented capabilities of the E and C++ programming languages, the access and operator methods (which are described in detail in Chapter 3) are implemented as iterator functions (25) using a recursive procedure; whenever a nested attribute is encountered in the plan tree, the procedure is recursively called. The use of recursion greatly simplifies the design and permits any level of nesting in the query.

The E code generator implements the access and operator methods by setting up an iterate loop that calls the operator methods. The operator and access methods are tested using sample queries on various database schemas to ensure the prototype database performs correctly.

The Triton system is used to store data on Structured Analysis (SA) diagrams (22) using the NRM representation as well as a relational representation of the same data. The schemas used are based on the work of Captain Gerald Morris (23). The performance of the NRM representation is compared to the relational representation; in particular, the advantage of the NRM over the relational model is demonstrated with respect to code generation time and query execution time.

All code produced in this research effort is documented in accordance with Air Force Institute of Technology system development documentation guidelines and standards (15).

1.6 Materials and Equipment

This research effort utilized the EXODUS facilities and tools on a Sun 3 workstation. The developers of EXODUS at the University of Wisconsin released EXODUS software modifications which were implemented at the Air Force Institute of Technology.

1.7 Outline of this Document

Chapter 2 describes the historical development of the NRM and presents significant work in NRM theory related to this research effort. This chapter also presents the EXODUS extensible database system. Chapter 3 begins with a discussion of the E programming language representation of nested relations. The chapter continues with a description of Triton's operator and access methods, including a presentation of the E code generator that implements these methods. Chapter 4 evaluates the performance of the NRM versus the relational model using the access and operator methods and discusses the advantages and disadvantages of using the EXODUS toolkit in the development of the Triton system. Chapter 5 provides a summary of this work and suggests areas for further research.

II. Overview of the Nested Relational Model and the EXODUS Extensible Database System

2.1 Overview

This chapter provides an overview of key research in the area of nested relational database systems as related to the development of the Triton nested relational database system. In the sections that follow, the motivation for the development of the nested relational model is discussed, followed by a brief summary of its historical development. Notable advances in nested relational database theory are also presented, including a brief survey of nested relational database implementations. Finally, the chapter concludes with a description of the EXODUS extensible database system, and how EXODUS is used in the development of Triton.

2.2 Impetus for the Development of the Nested Relational Model

The development of database management systems has been driven by the changing needs of their users. The requirement for fast access to reliable data by multiple users sparked the creation of database management systems in the 1960's. For the next twenty years, the hierarchical and network models dominated the field of database design. In 1970, Codd (8) introduced a data model based on a tabular format, called the relational model. However, widespread use of the relational model did not occur until the late 1970's when straightforward query languages for the relational model were developed and prototype systems validated the model's efficiency (30:2).

Today, the major push in the realm of database research comes from a demand for non-standard or non-business applications. These applications include engineering design, such as computer-aided design (CAD) and computer-aided software engineering (CASE), as well as office automation (33:2). These new applications require more complex data models to efficiently map hierarchical data. "A database model should allow databases to be viewed in a manner that is based upon the meaning of data as seen by its users..." (4:xvii). The nested relational model is an attempt to represent complex data within a relational model framework.

In the nested relational model, as with the relational model, information is stored in a tabular format. However, attributes in the relation do not have to be atomic, but can have set-valued or relation-valued attributes. This allows information to be stored in the database in a way that corresponds to the user's interpretation of the data. Traditional relational systems break complex data into several first normal form relations. Since non-standard applications involve queries that require access to many of these relations at once, computationally expensive joins are needed to correlate all the necessary data. Nested relations require fewer joins for complex data since the data is spread across fewer relations, making the data model less confusing to users and database processing more efficient.

2.3 Historical Development of the Nested Relational Model

2.3.1 The Relational Model. The nested relational model is the descendant of the relational model introduced by Codd (8). Codd's seminal work represents data as tables, called *relations*. Data in the relations are accessed via a *key* that uniquely indentifies each row, or *tuple*, of the relation. Figure 2.1 shows a simple relational database with five relations that hold information on VHSIC Hardware Description Language (VHDL) designs (1).[1] In the relational model, all attributes of every relation must be atomic, or non-decomposable. In other words, attributes may contain only one character string, one integer, or one real number. When a relation conforms to this constraint, it is in first normal form (1NF).

2.3.2 Introduction of Set-Valued Attributes. Makinouchi (20) first suggested removing the 1NF requirement, laying the groundwork for the nested relational model. A model allowing attributes to have sets of values (called set-valued attributes) was introduced by Jaeshke and Schek (17). Figure 2.2 shows the relational database for the VHDL example used in Figure 2.1 where set-valued attributes are allowed. Notice that the only relation affected by the change is the *SYS_TO_COMP* relation, where the value for the *COMP#* attribute is a set of component numbers. Using set-valued attributes limits the size of the database because *SYS#* does not have to be repeated for every occurance of *COMP#*.

[1]VHSIC stands for *very high speed integrated circuits.*

SYSTEMS

SYS#	NAME
43191	COUNTER
14701	FULL_ADD

COMPONENTS

COMP#	NAME
15899	CLOCK_GEN
30018	CNTRL_CTR
41572	MAJORITY
81909	XOR_GATE

SYS_TO_COMP

SYS#	COMP#
43191	15899
43191	30018
14701	41572
14701	81909

COMPONENT_PORTS

COMP#	NAME	MOD	TYPE	STRT BIT	STP BIT
15899	RUN	in	BIT	0	0
15899	CLK	out	BIT	0	0
30018	CLK	in	BIT	0	0
30018	STRB	in	BIT	0	0
30018	CON	in	BIT_V	0	1
30018	DATA	in	BIT_V	0	3
30018	COUT	out	BIT_V	0	3
41572	A	in	BIT	0	0
41572	B	in	BIT	0	0
41572	C	in	BIT	0	0
41572	MAJ	out	BIT	0	0
81909	A	in	BIT	0	0
81909	B	in	BIT	0	0
81909	C	out	BIT	0	0

SYSTEM_PORTS

SYS#	NAME	MOD	TYPE	STRT BIT	STP BIT
43191	STRT	in	BIT	0	0
43191	STROB	in	BIT	0	0
43191	CON	in	BIT_V	0	1
43191	DATA_B	in	BIT_V	0	3
43191	CNT	out	BIT_V	0	3
14701	X	in	BIT	0	0
14701	Y	in	BIT	0	0
14701	CIN	in	BIT	0	0
14701	Z	out	BIT	0	0
14701	COUT	out	BIT	0	0

Figure 2.1. Representation of a Relational VHDL Database

2.3.3 Introduction of Relation-Valued Attributes. Thomas and Fischer (32) went another step further, suggesting that attributes of a relation be allowed to hold not only sets of values, but complete relations. A database that allows relation-valued attributes is called a *nested relational database*. Figure 2.3 shows the nested relational database for the VHDL example. This database is composed of two nested relations with one level of nesting in each. The advantage of this model comes from the storage savings realized because the *SYS#* and *COMP#* attributes are not each repeated three times in the database schema (as they are in the two previous database schemas). In addition, only one join is required to correlate all data in the nested database, as compared to the four joins required in each of the two previous schemas.

SYSTEMS

SYS#	NAME
43191	COUNTER
14701	FULL_ADD

COMPONENTS

COMP#	NAME
15899	CLOCK_GEN
30018	CNTRL_CTR
41572	MAJORITY
81909	XOR_GATE

SYS_TO_COMF

SYS#	COMP#
43191	{15899, 30016}
14701	{41572, 81909}

COMPONENT_PORTS

COMP#	NAME	MOD	TYPE	STRT BIT	STP BIT
15899	RUN	in	BIT	0	0
15899	CLK	out	BIT	0	0
30018	CLK	in	BIT	0	0
30018	STRB	in	BIT	0	0
30018	CON	in	BIT_V	0	1
30018	DATA	in	BIT_V	0	3
30018	COUT	out	BIT_V	0	3
41572	A	in	BIT	0	0
41572	B	in	BIT	0	0
41572	C	in	BIT	0	0
41572	MAJ	out	BIT	0	0
81909	A	in	BIT	0	0
81909	B	in	BIT	0	0
81909	C	out	BIT	0	0

SYSTEM_PORTS

SYS#	NAME	MOD	TYPE	STRT BIT	STP BIT
43191	STRT	in	BIT	0	0
43191	STROB	in	BIT	0	0
43191	CON	in	BIT_V	0	1
43191	DATA_B	in	BIT_V	0	3
43191	CNT	out	BIT_V	0	3
14701	X	in	BIT	0	0
14701	Y	in	BIT	0	0
14701	CIN	in	BIT	0	0
14701	Z	out	BIT	0	0
14701	COUT	out	BIT	0	0

Figure 2.2. Representation of a VHDL Database with Set-Valued Attributes

SYSTEMS

SYS#	NAME	COMP#	PORTS				
			NAME	MOD	TYPE	STRT BIT	STP BIT
43191	COUNTER	15899	STRT	in	BIT	0	0
			STROB	in	BIT	0	0
			CON	in	BIT_V	0	1
		30018	DATA_B	in	BIT_V	0	3
			CNT	out	BIT_V	0	3
14701	FULL_ADD	41572	X	in	BIT	0	0
			Y	in	BIT	0	0
			CIN	in	BIT	0	0
		81909	Z	out	BIT	0	0
			COUT	out	BIT	0	0

COMPONENTS

COMP#	NAME	PORTS				
		NAME	MOD	TYPE	STRT BIT	STP BIT
15899	CLOCK_GEN	RUN	in	BIT	0	0
		CLK	out	BIT	0	0
30018	CNTRL_CTR	CLK	in	BIT	0	0
		STRB	in	BIT	0	0
		CON	in	BIT_V	0	1
		DATA	in	BIT_V	0	3
		COUT	out	BIT_V	0	3
41572	MAJORITY	A	in	BIT	0	0
		B	in	BIT	0	0
		C	in	BIT	0	0
		MAJ	out	BIT	0	0
81909	XOR_GATE	A	in	BIT	0	0
		B	in	BIT	0	0
		C	out	BIT	0	0

Figure 2.3. Representation of a VHDL Database with Relation-Valued Attributes

2.4 Notable Advances in Nested Relational Database Theory

Following the development of the nested relational model (NRM), research in the areas of query languages, nested relational algebra and its optimization, as well as specialized access methods have demonstrated the viabilty of the NRM. The next three sections review the work in each of these three areas relevant to the development of the Triton nested relational database system.

2.4.1 The SQL/NF Query Language. In 1987, Roth, Korth, and Batory (26) developed an extended version of the SQL query language for use on nested relations, called SQL/NF. As with SQL, SQL/NF uses SELECT-FROM-WHERE (SFW) expressions to pose queries involving any level of nesting in a nested relation. Unlike SQL, SQL/NF allows nested SFW expressions in both the SELECT and FROM clauses to manipulate relation-valued attributes. Using the nested relational schema of Figure 2.3, an SQL/NF query to retrieve the system name and port names for system 14701 is

```
SELECT name, (SELECT name
              FROM ports)
FROM systems
WHERE sys# = 14701
```

The outer SFW-expression retrieves the name of the system where the system number is 14701. The inner (nested) SFW-expression selects the name of each port of system 14701.

An SQL/NF query to retrieve the component names and component port names for the components of system 14701 is

```
SELECT components.name, (SELECT name
                         FROM components.ports)
FROM systems, components
WHERE systems.comp# = components.comp# AND sys# = 14701
```

The Cartesian product of the two relations, *systems* and *components* is formed, and the only tuples selected are those where the *comp#* attribute of *systems* and *components* is the same and *sys#* is 14701. Then the *name* attributes of *components* and *ports* is projected out.

These simple examples do not completely demonstrate the full power of SQL/NF. However, they do show how complex queries can be simply posed to a nested relational database. Because of its simplicity and understandability, SQL/NF was chosen as the query language for the Triton system (28).

2.4.2 Colby Algebra and its Optimization. While query languages, such as SQL/NF, are intended to provide an easy query capability for database users, rigid mathematical algebras have been adopted for use on the relational and nested relational models to formalize the representation of queries. Natural language queries are usually translated into the more rigorous algebra before being processed. The first algebras developed for the NRM (32) required nested relations to be restructured to a relational format before information could be extracted from the database. However, Colby (9) suggested a recursive algebra that retrieves information from any level of nesting in a relation without first restructuring the relation.

Colby redefined the *select* (σ), *project* (π), and *join* ($\bowtie$) operators of the relational model, and introduced two restructuring operators *nest* (ν) and *unnest* (μ). She also decribed optimization techniques for her recursive algebra. Appendix A presents the operation of the Colby algebra by some examples. Because of the ease with which SQL/NF is mapped into the Colby algebra, the Colby algebra was selected for the Triton system.

2.4.3 Indexing Techniques and Access Methods. In databases, it is sometimes useful to have an index on an attribute or several indices on a number of attributes to facilitate quick access to a particular value. In the next two subsections, two papers are reviewed that discuss indexing techniques and their use as access methods for nested relational databases. In the first article, Bertino and Kim (3) present three indexing techniques for use on nested relations. In the second article, the developers of the ANDA nested database system (11) discuss a unique indexing structure and its use for efficient access to information in their database. At the present time, no indexing techniques have been implemented for the Triton system. As such, these two articles are intended to serve as a recommended approach for future developers of the Triton system.

2.4.3.1 Bertino and Kim's Indexing Techniques. The first indexing technique introduced by Bertino and Kim sets up a *nested index*, that correlates the value of an atomic attribute at some level of nesting with the indices of those tuples at the outermost level of the relation that contain that value. The second index type, called the *path index* correlates the value of an atomic attribute at some level of nesting with a set of path indices along every level of nesting correponding to those tuples with that value. The third and final index type introduced by Bertino and Kim is the *multiindex* which, given a specific path, creates a separate index for each subpath.

Following the presentation of the three indexing techniques, Bertino and Kim compared them based on storage cost, retrieval cost, and update cost. They concluded that the nested index has the lowest storage cost and the best retrieval performance, while the multiindex is best with respect to update performance.

2.4.3.2 ANDA Nested Database System. Deshpande and Van Gucht (11) implemented a nested relational database called ANDA. Of particlular interest to this research are the access methods and mechanisms ANDA uses for the retrieval of data from the database. ANDA makes a distinction between value-driven operations, such as select, join, and nest, and structure-oriented operations, such as project and unnest.

Speed of retrieval of information is maximized for both types of operations by using two different storage structures; VALTREE is the value-driven indexing structure, and RECLIST is a special record-list structure used for structure-oriented operations. VALTREE maps values to a list of tuple identifiers in all relations and relation-values attributes that contain that value. RECLIST maps these tuple identifiers to the actual physical addresses where these tuples are stored. For increase efficiency, ANDA stores tuple identifiers obtained from VALTREE in a special cache consisting of a set of stacks. Once these tuple identifiers have been processed within the cache as specified by the query, RECLIST retrieves the actual values or tuples in response to the query.

2.5 Related Work

This section is intended to provide a brief survey of nested relational database implementations. This section describes the Advanced Information Management Prototype, the Darmstadt Database System, and the Verso DBMS, focusing on their storage management techniques and access methods.

2.5.1 Advanced Information Management Prototype (AIM-P). The Advanced Information Management Prototype (AIM-P) (10) was developed by IBM as a research vehicle in the realm of non-standard applications. There are two similarities between AIM-P and the Triton system: (1) both are intended to be the database implementation "back-end" for design application tools, and (2) both use the nested relational data model to represent the underlying structure of the database.

Because AIM-P was developed completely from scratch, the developers had the freedom to implement their design to take full advantage of the underlying data model (as opposed to the Triton system, which was constrained by the specific capabilities of the EXODUS storage manager).[2] Thus, AIM-P's physical storage structure differs from Triton's in that Triton maps relation definitions into a programming language, while AIM-P uses a tree structure to hold the same information. The advantage of a tree structure representation of data is that tuples can be added or deleted quickly at any level of nesting by simply adding or deleting a data pointer to the appropriate internal node of the tree.

2.5.2 Darmstadt Database System (DASDBS). Developed at the Technical University of Darmstadt, the Darmstadt Database System (DASDBS) (16, 27) is "...a *kernel* that integrates the common features of a rather low-level storage component, but allows efficient and flexible *front ends* tailored to specific application classes..." (27:51).

The DASDBS kernel provides access (such as reading, insertion, and deletion) to sets of complex objects as opposed to a one-record-at-a-time interface by fetching or storing lists of pages via a variable size buffer. Thus, a single scan of a complex object retrieves all of the values of its sub-objects, which limits the number of disk accesses. This is very similar

[2]This is explained in more detail in Chapter 4.

to the way the EXODUS storage manager works (5). The kernel provides operations to read an object (similar to EXODUS's *scan*), insert an object (similar to EXODUS's *in ... new* construct), and delete an object (similar to EXODUS's *delete*). Like the EXODUS storage manager, the DASDBS kernel proviαes concurrency control capabilities.

2.5.3 The Verso DBMS. The Verso DBMS (29) is a relational database system that stores data in nested form (called *V-relations*) to increase query processing speed. The Verso system consists of three layers:

1. The V-relation level, which is made up of V-relations and their schema
2. The file level, which is made up of the index and the physical representations of V-relations
3. The lowest level, which is made up of blocks

This is similar to the Triton system, in that the V-relation levelcorresponds to Triton's system catalogs, the file level corresponds to the E language representation of relations, and the lowest level corresponds to the EXODUS storage manager.

The Verso system utilizes a "filter" (mapper) for on-the-fly processing of tuples, which can perform all algebraic operations except for restructuring actions. The filter is implemented as a finite state automaton (FSA), which scans the V-relation one byte at a time. The advantage of using such a filter is that query processing is much faster because it is mapped to a very low-level representation of the problem space on a dedicated machine. The disadvantage is that this low-level representation is complex and difficult to grasp, making the filter hard to modify and maintain.

2.6 EXODUS Extensible Database System

The creation of a new database system is not a trivial task. As with any major software system, significant time is spent in the design, implementation, and testing phases. Because of the need for the development of new database systems with novel capabilities, extensible database systems have been designed to simplify the production of application-specific DBMSs. Two such systems are the EXODUS extensible database system, developed at the University of Wisconsin (7), and the GENESIS system (2). The EXODUS system is not a DBMS, but a toolkit that provides the necessary facilities and tools to aid in the development of new DBMSs. The goal of the EXODUS system is the provision of extensibility without sacrificing performance (7:475).

Because of its power, flexibility, and availability, EXODUS was selected to aid in the development of the Triton system. EXODUS provides generic system components and furnishes component *generators* to aid in the construction of DBMS-specific components. When neither approach is possible, EXODUS provides tools to aid in the development of the component. The EXODUS tools used in the development of Triton include:

- the storage manager (5, 6), which stores the physical data of the database and provides access to the data via procedural calls
- the E persistent programming language and its compiler (25, 24)

2.6.1 EXODUS Tools Used in the Production of the Triton Nested Database System. The architecture of the Triton system is given in Figure 2.4. At the present time, Triton's parser (28) is able to parse all possible SQL/NF statements, but only builds query trees for the statements that create and delete items in the system catalogs, as well as the statements that directly translate into the appropriate algebraic operations of select, project, and Cartesian product. Triton's parser component was implemented using the UNIX tools of YACC and LEX.

The query tree built by the parser is passed to the rule-based optimizer where it is changed to a plan tree. In the optimizer, the sequence and contents of the nodes in the query tree are changed into an optimized plan tree and relational operators are replaced with specific operator methods. At the present time, Triton's optimizer component has not been developed. However, the intention is to use the EXODUS optimizer generator (13, 12) to generate this component. The EXODUS optimizer generator takes as input (1) a set of operators, (2) a set of methods that implement the operators, (3) transformation rules that describe equivalence-preserving transformations of query trees, and (4) implementation rules that describe how to replace an operator with a specific method. Using these rules, a specific optimizer is generated for the particular application.

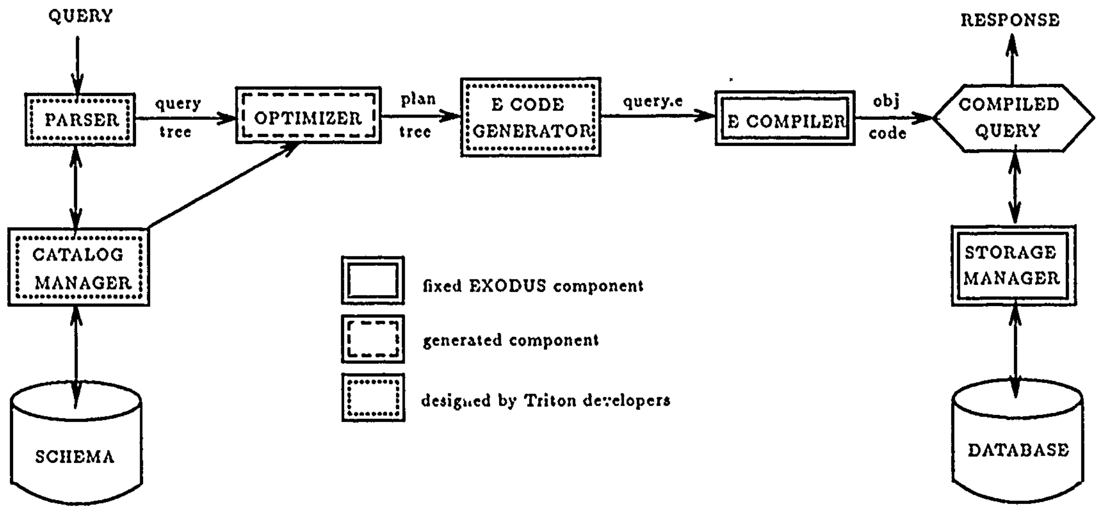

Figure 2.4. The Triton Nested Database System Architecture Using EXODUS

We chose to use the EXODUS optimizer generator for Triton because the relational algebra used by Triton lends itself to EXODUS' rule-based method. This modular approach to database development will reduce the amount of code required for implementation of the Triton system. The only unique code Triton's developers will need to write will be the additional functions that are called by the optimizer when implementing a specific operator or access method. With accurate cost functions, we anticipate the generated optimizer will work as well as a custom built one.

Using the programming constructs provided by the E programming language, the E code generator is designed to traverse the plan tree and generate the E code to effect the operator and access methods to perform the query. The EXODUS E compiler is used to link the methods with the E code that specifies the relation definitions.

The storage manager is a fixed component provided by EXODUS. Conceptually, the storage manager is the layer between the access and operator methods and the physical data in the database. The storage manager is accessed via procedural calls in the compiled query which allow the creation, destruction, and iteration through the contents of database files. Objects can be inserted in and deleted from a file at any offset in the file, and explicit clustering of objects on disk can be specified. The storage manager provides procedures for transaction management as well as versioning of objects.

2.7 Summary

The goal of this chapter was to familiarize the reader with several key topics related to this research. This chapter introduced the nested relational model and described its historical development. Notable advances in nested relational theory relevant to the Triton system were described, including a brief survey of nested relational database implementations. Finally, the EXODUS extensible database system was described, focusing on how the EXODUS toolkit was used to develop the Triton system. The following chapter describes in more detail how EXODUS is used to develop the access and operator methods for the Triton system.

III. Design and Implementation

3.1 Overview

The goal of this research is to design and implement operator and access methods for the Triton nested relational database system to efficiently process queries on nested data. First, operator methods are developed to implement the project, select, and join operators in the Colby relational algebra. Second, access methods are implemented to add a relation, delete a relation, store tuples, delete tuples, and modify tuples. The methods are implemented by the E code generator called *codegen*. Figure 3.1 shows how the E code generator fits into the backend of the prototype database.

The optimizer produces a plan tree, which structurally represents the database user's query. The nodes of the plan tree contain specific operator and access methods. The program *codegen* traverses the plan tree in postorder and, through the use of templates, generates the code required to implement the query. This code is written to a file called *query.e*, which, when executed, performs the query. A unique *query.e* file is dynamically created for each user query. If a relation is added to or deleted from the database, *codegen* does not create the file *query.e*, but directly performs the necessary actions to add or delete the relation.

The remainder of this chapter is divided into three sections. The first section describes how nested relations are represented in Triton. The second section describes Triton's operator and access methods. Finally, the third section discusses the design of the E code generator that implements the methods.

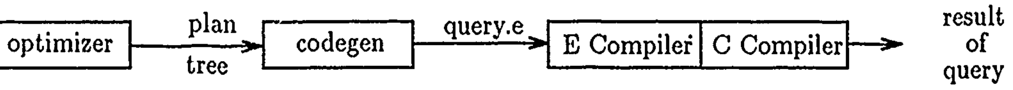

Figure 3.1. Prototype Nested Database Backend

3.2 Representation of Nested Relations in Triton

Because of its central importance to the Triton system, it is important to know how nested relations are represented in E. A brief discussion of the declaration and representation of both non-nested and nested relations in the E programming language follows.

3.2.1 E Representation of Non-Nested Relations. Figure 3.3 shows the relation specification for the non-nested *children* relation of Figure 3.2. A tuple of the *children* relation is of type *child*. The attributes are specified before the keyword **public**. Following the keyword **public** are the constructor and member functions. The constructor function takes as input three character pointers and an integer to initialize a tuple in the *children* relation. The *get_emp_ssn*, *get_child_name*, *get_child_age*, and *get_sex* member functions return the value of the corresponding attribute. The *change_emp_ssn, change_child_name, change_child_age*, and *change_child_sex* member functions enable the values of the attributes to be changed. The print member function takes a pointer to a child tuple and prints out the values of the attributes. The implementation of the member functions are not given here. The line

```
dbclass child_relation: collection [child];
```

specifies that a *child_relation* type consists of a collection of tuples of type *child*. The line

```
persistent child_relation children;
```

declares a persistent relation called *children*, which is of type *child_relation*.

3.2.2 E Representation of Nested Relations. The E programming language supports the implementation of nested relations through the use of collections for relation-valued attributes. Figure 3.5 shows the E language relation specification for the *employee* relation of Figure 3.4. The *emp* type contains four atomic attributes called *dept*, *emp_name*, *emp_age*, and *emp_ssn* as well as one relation-valued attribute called *children*, which is a collection of tuples of type *child*. As a requirement, only atomic attributes are specified in the constructor and member functions. The *employee* relation is a persistent

emp_ssn	*child_name*	*child_age*	*sex*
237-46-3567	Jeramie	8	M
237-46-3567	Todd	4	M
395-73-8901	Susan	3	F
555-12-3434	Laura	18	F
555-12-3434	John	13	M
555-12-3434	Matthew	11	M
624-35-8152	George	5	M
624-35-8152	Janis	3	F

Figure 3.2. The *Children* Relation

object of type *emp_relation* made up of a collection of tuples of type *emp*. Any level of nesting can be represented in the E programming language in a similar fashion.

```
dbstruct child {
    dbchar emp_ssn[12];
    dbchar child_name[32];
    dbint child_age;
    dbchar sex[2];
public:
    child (char *, char *, int, char *);
    char * get_emp_ssn();
    void change_emp_ssn (char *);
    char * get_child_name();
    void change_child_name (char *);
    int get_child_age();
    void change_child_age (int);
    char * get_sex();
    void change_sex (char *);
    void print (child *);
};

dbclass child_relation: collection [child];
persistent child_relation children;
```

Figure 3.3. E Specification of the *Children* Relation

dept	*emp_name*	*emp_age*	*emp_ssn*	*children*		
				child_name	*child_age*	*sex*
Mktg	J. Smith	27	237-40-3567	Jeramie	8	M
				Todd	4	M
Rsrch	M. Taylor	31	395-73-8901	Susan	3	F
Adv	T. Therrien	43	555-12-3434	Laura	18	F
				John	13	M
				Matthew	12	M
Pers	C. Dunlap	37	624-35-8152	George	5	M
				Janis	3	F

Figure 3.4. The Nested *Employee* Relation

```
dbstruct child {
    dbchar child_name[32];
    dbint child_age;
    dbchar sex[2];
public:
    child (char *, int, char *);
    char * get_child_name();
    void change_child_name (char *);
    int get_child_age();
    void change_child_age (int);
    char * get_sex();
    void change_sex (char *);
    void print (child *);
};

dbstruct emp {
    dbchar dept[20];
    dbchar emp_name[32];
    dbint emp_age;
    dbchar emp_ssn[12];
    dbclass childRVA: collection [child];
    childRVA children;
public:
    emp (char *, char *, int, char *);
    char * get_dept();
    void change_dept (char *);
    char * get_emp_name();
    void change_emp_name (char *);
    int get_emp_age();
    void change_emp_age (int);
    char * get_emp_ssn();
    void change_emp_ssn (char *);
    void print (emp *);
};

dbclass emp_relation: collection [emp];
persistent emp_relation employee;
```

Figure 3.5. E Specification of the *Employee* Relation

3.3 Design and Function of Methods

In this research effort, I designed and implemented two operator methods. The *filescan* method can perform a select, a project, or both a select and project. The *loops_join* method implements the join operator and can perform a select and/or project in conjunction with the join. In addition, five methods were implemented that (1) create a relation, (2) delete a relation, (3) add tuples to a relation, (4) modify tuples in a relation, and (5) delete tuples from a relation. The methods that create and delete a relation are not generated by *codegen* and written to the *query.e* file. Instead, *codegen* directly performs the operations necessary to create or delete the relation. Because of this, the create relation and delete relation methods are not described in this section, but are explained in Section 3.4.

The design and implementation of my operator and access methods utilize several data structures. Because these structures play a vital role in understanding the design of Triton's operator and access methods, Appendix B explains each one in detail.

3.3.1 The Filescan Method.

The *filescan* method is implemented as an iterator function (25) that yields tuples to the *iterate* statement with the schema defined by the projection list. If a projection is taking place, the *filescan* method first defines a temporary nested relational schema to hold a tuple of the projected input relation. Once this temporary relation template has been defined, *filescan* iterates through each tuple of the input relation and determines if the tuple meets the selection condition, if such a condition exists. If the tuple meets the selection condition, the appropriate attributes are copied into the temporary relation template. A pointer to this temporary relation tuple is yielded (returned) to the calling procedure.

3.3.1.1 Plan Tree Structural Representation of a Filescan.

Using the nested relational schema for the *employees* nested relation contained in Appendix C, the following example illustrates how the filescan method works. Assume we have a requirement to do a project and a select on the *employees* relation. The attributes to be projected out from *employees* are *name* and *age*, and from the *children* relation-valued attribute of *employees*, the *name* and *age* attributes must be projected. From the *toys* relation-valued attribute of *children*, the *name* attribute must be projected. The selection condition extracts only those tuples where the employees' age is over 30 and the children's age is less than 5.

The Colby algebra equivalent of this query is:

$$\pi\ ((\text{name, age, children (name, age, toys (name)))}\ \sigma\ (\text{employees}_{age>30}\ (\text{children}_{age<5})))$$

The plan tree representation of this query is shown in Figure 3.6. Notice that since the predicate, $age < 30$, pertains to the atomic attributes at the highest level of the *employees* relation, the predicate tree hangs off the *pred* portion of the *plan* node. Since the second predicate, $age < 5$, pertains to the *children* relation-valued attribute, the one node predicate tree hangs off the *cond* portion of the *children* list node. At first, I attempted to make one predicate tree that specified all conditions for the query regardless of what level of nesting they pertained to, but found this required many traversals of the entire predicate tree to extract conditions (one traversal for each level of nesting). I found it was more efficient (in terms of time) to separate the predicate into smaller predicate trees, one for each level of nesting. This way, only the conditions relevant to that level are present, and the predicate tree is not unnecessarily traversed if no conditions exist for that level.

Each projected attribute is represented by a *list* node. Relation-valued attributes also appear as *list* nodes, but only *list* nodes representing relation-valued attributes can have values for *cond* and *sublist*. The *cond* field of the *list* node for the relation-valued attribute *children* points to the one node predicate tree representing the condition $age < 5$. Since there are attributes being projected from the relation-valued attribute *children*, the *sublist* field of the *list* node representing the *children* attribute points to another linked list of *list* nodes representing those projected attributes of *children*. I thought of representing the projected attributes at all levels of nesting as one linked list, rather than using sublists for the projected attributes of relation-valued attributes. However, the use of sublists give the representation of nested relations a recursive appearance, which lends itself well to the use of recursion in the construction of the operator and access methods by the code generator *codegen*.

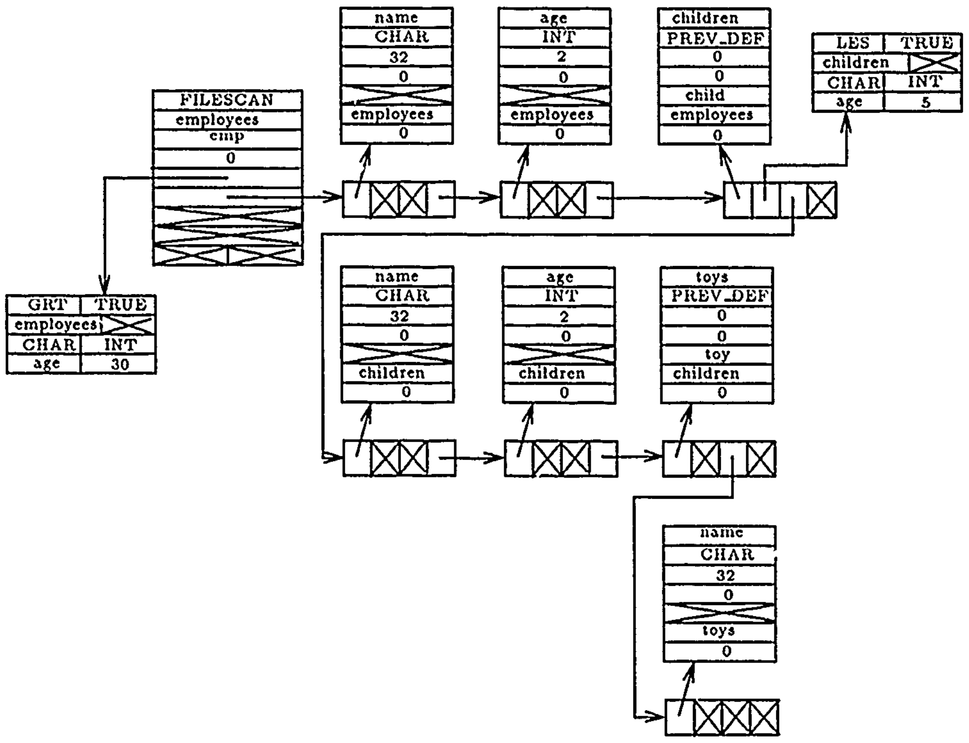

Figure 3.6. Example Plan Tree for *Filescan* Query

3.3.1.2 How the Filescan Method Works. The code that implements the *filescan* for the tree discussed above is generated by *codegen* and contained in the file *query.e*. The contents of *query.e* for this plan tree are given in Appendix E. The code between the **extern** statement and the **iterator** statement is the temporary relation schema that specifies the structure of a relation with the necessary attributes projected out from the input relation. This specification also includes the code to implement the constructor and member functions of the temporary relation.

The iterator actually performs the *filescan*. The *filescan* iterator is set up as a series of steps for each level of nesting. These steps are:

1. iterate through the input relation, or relation valued attribute of the input relation, scanning one tuple at a time
2. test the tuple to see if it meets the selection criterion, if one exists for that level of nesting

3. project the necessary attributes from the input tuple (by using the *get_* member functions) and record them in the temporary relation schema (by using the constructor function)
4. set up necessary reference pointers to the input relation and to the temporary relation if there is another level of nesting in the input relation

At the outermost level of nesting, after all other relation-valued attributes have been processed, a pointer to the temporary tuple is returned by the *filescan* iterator. The main program of *query.e* iterates through the *filescan* tuple-by-tuple and prints out the values of the attributes.

3.3.1.3 Rationale Behind Design Decisions for the Filescan Method. I did not initially implement the *filescan* method as an iterator function, but performed the projection and selection in the main program along with the code that prints the result. I decided to design the filescan as a separate function to make the code in the *query.e* file modular and less confusing, which is particularly useful when the *query.e* file implements several filescans in one query (which occurs when filescans filter tuples to a join, as will be discussed in Section 3.3.2).

After I decided to make the *filescan* a separate function, I designed it to return the resultant relation as a whole after the processing was complete. However, it is much faster to filter the relation a tuple at a time if further processing of the tuple is going to take place (for example, if the filescan is filtering tuples to a join). This way, the filescanned relation is only processed once. I chose to implement the *filescan* as an iterator function, because iterator functions are controlled looping constructs that automatically step through a sequence of items. Since each tuple of the filescanned relation needs to be processed further (for a join) or printed (for a filescan alone), I felt that designing the *filescan* method as an iterator function would elegantly serve my needs.

3.3.2 The Loops_Join Method. The *loops_join* method is implemented as an iterator function that yields tuples with the schema defined by the join. The *loops_join* method can implement a projection and/or a selection in conjunction with the join. The *loops_join* method first defines a temporary nested relational schema to hold a tuple of the result of the join, or a tuple of the projected result of the join (if a projection is also taking place). The *loops_join* method iterates through each tuple of the left input, and for each tuple, it iterates through all tuples of the right input. If the tuples meet the join criterion, the appropriate attributes of each are copied into the temporary relation. If no join criterion is specified, the *loops_join* method acts as a cross product. A pointer to the temporary relation tuple is returned to the calling procedure.

3.3.2.1 Plan Tree Structural Representation of a Loops_Join. To illustrate how the *loops_join* method works, the nested relation *employees* in Appendix C and the nested relation *products* in Appendix D are used. The required operation is to join the *products* relation with the *toys* relation-valued attribute of *children*, where *children* is a relation-valued attribute of *employees*. The join criterion is that the name of the product must be the same as the name of the toy. To make this example more realistic, assume a *filescan* is going to take place on *employees* that projects out the employee's name as well as the name of the toy. In addition, assume a *filescan* is going to take place on *products* that projects out the product's name and all the information on the manufacturer of the product.

The Colby algebra equivalent of this query is:

$$\bowtie_\theta (\pi \text{ ((name, children (toys (name))) employees) (children (toys)), } \pi \text{ ((name, manufacturer) products))}$$

$$\text{where } \theta \text{ is toys.name = products.name}$$

The plan tree representation of this query is shown in Figure 3.7. The two inputs to the *loops_join* node are *filescan* nodes that each have attribute lists for their projections. The second attribute of the *products* filescan is a relation-valued attribute but does not have a sublist, hence all attributes of *manufacturers* are to be projected out. The *loops_join* node specifies the join criterion, which hangs off the *pred* field of the *plan* node. Notice that the *constant_on_right* field of the *pred* node is false, meaning that both operands in the predicate are attribute names (and not constants).

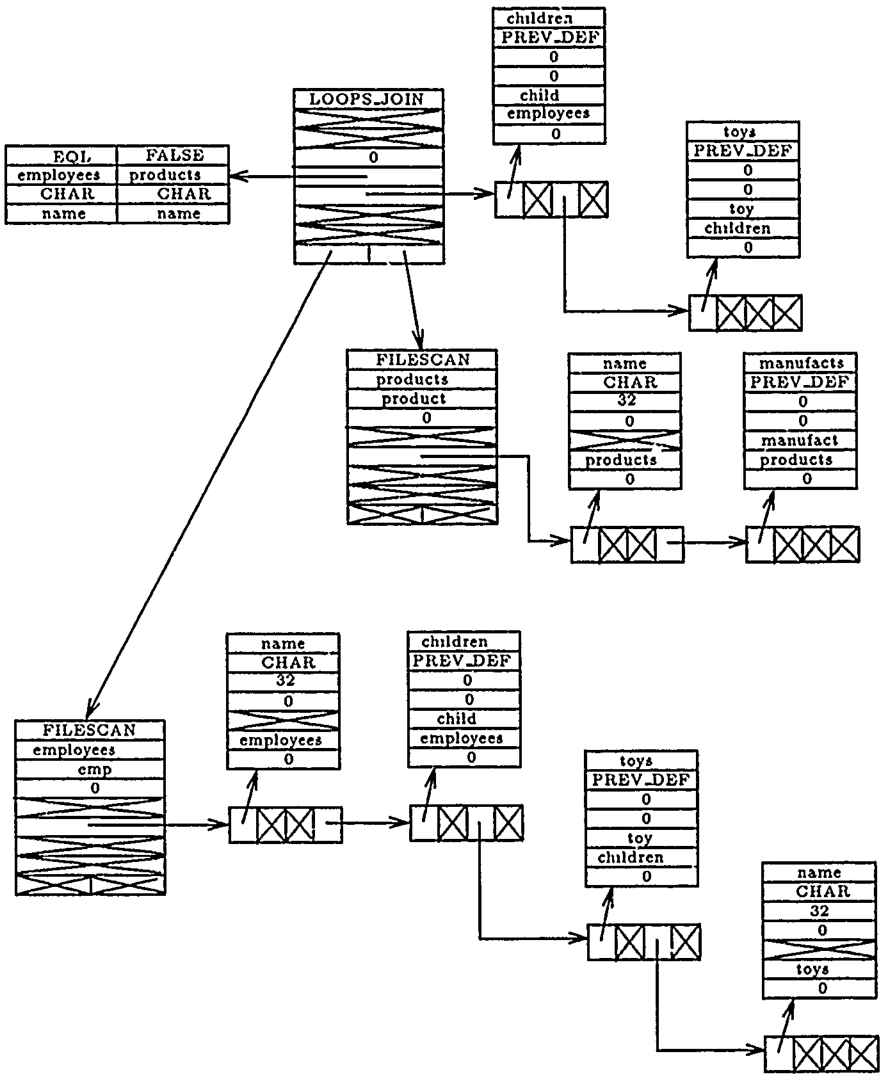

Figure 3.7. Example Plan Tree for *Loops_Join* Query

On the *loops_join* node, the Colby join path is specified by the two *list* nodes, *children* and *toys*. This means that the join is to occur between the *toys* relation-valued attribute and the *products* relation. While the Colby algebra enables a join to occur within a nested relation-valued attribute of the first relation, the join can only occur at the top level of the second relation. Otherwise, the joined schema would not be constructable.

3.3.2.2 How the Loops_Join Method Works. As with the *filescan* method, the code that implements the *loops_join* method for the plan tree shown in Figure 3.7 is generated by *codegen* and contained in the file *query.e*. Appendix F shows the contents of *query.e* for this plan tree. The group of code before the main program can be logically separated into three sections:

1. the templates and *filescan* iterator function for the *employees* relation
2. the templates and *filescan* iterator function for the *products* relation
3. the templates and *loops_join* iterator function for the join of the *products* relation to the *employees* relation

The first two sections (which implement the two filescans) operate as discussed in Section 3.3.1. The third section of code (which implements the join) is similar to the two filescan sections, but has some notable differences.

First, a temporary relation structure is defined that holds a tuple of the result of the join. This is performed the same way as in the *filescan* method. The *temp5* structure is the level where the join is occuring. Notice that the *name* attribute of *toys* has been renamed to *toys_name* and the *name* attribute of *products* has been renamed to *products_name*. The reason for the renaming is that attribute names must be unique within a level of nesting. Since the joining of the relations would put two occurances of *name* at one level of nesting, these two attributes are given a unique name by *codegen* by combining the parent relation name with the attribute name.

The *loops_join* iterator performs the join. For each tuple of the filescan on the left, if the join level has not been reached, the join template is filled. This continues until the join level is reached. Then for each tuple of the filescan on the right, if the join condition is satisfied, the join template is filled with information from the two relations. If there are any relation-valued attributes present in the join template at this level, reference pointers are set up to descend a level of nesting. All relation-valued attributes that belong to the left and right relations are filled. Finally, after a single tuple of the join template has been filled, a pointer to this tuple is yielded (returned) by the *loops_join* iterator.

The main program of *query.e* uses the *loops_join* iterator to step through each tuple of the joined relation and prints out the value of each attribute.

3.3.2.3 Rationale Behind Design Decisions for the Loops_Join Method. Because the *loops_join* method may feed a join above it, I also implemented this method as an iterator function. The alternative is to return the joined relation after processing as a whole; however, if further processing is taking place above the join, it is more efficient in terms of time to perform the join as a filter, passing tuples one at a time to the node above. I designed the *loops_join* method to carry out a projection and selection in conjunction with the join to avoid having to process all the tuples of the join a second time.

3.3.3 Store_Values Method. The *store_values* method can insert any number of tuples into a nested relation at any level of nesting. Condition statements are used to locate the correct tuple in the relation where the new values are to be stored. The code generator creates the *query.e* file which contains the code to implement the *store_values* method.

As an example, assume the *employees* schema as defined in Appendix C is used. Suppose we wish to add a new project to the employee David and two new toys to his daughter, Flo. Figure 3.8 gives the plan tree representation of this query. The leaves of the tree are values that are to be inserted into the relation. At the leaf level, each attribute of the relation or relation-valued attribute has, by definition, a corresponding *list* node containing the value of the attribute for this new tuple. This value is contained in the *value* field of the *attrdesc* node. If a relation-valued attribute is being into the relation, it may have a sublist pointing to the values of its attributes. A null sublist indicates that there are no tuples for this relation-valued attribute.

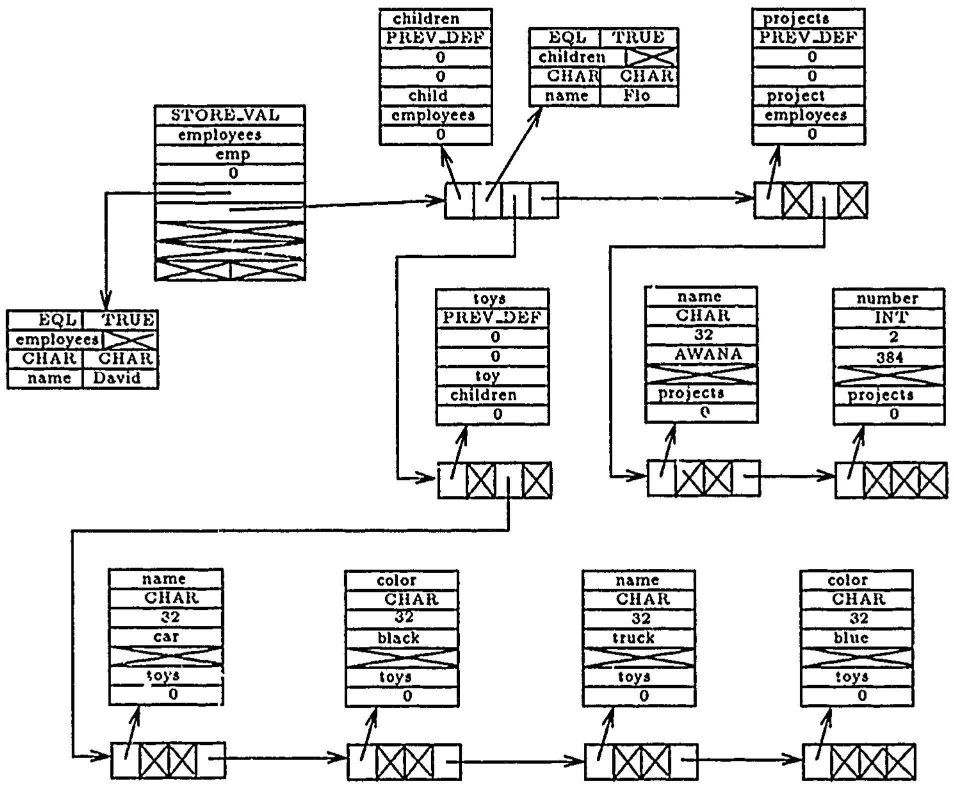

Figure 3.8. Example Plan Tree For *Store_Values* Query

The contents of *query.e* for the plan tree of Figure 3.8 is given in Appendix G. The method scans each tuple of *employees* until David is reached. Then David's children are scanned until Flo is reached, where two new toys are added. Finally, a new project is added to David's *projects* relation-valued attribute.

3.3.4 Modify Method. The *modify* method can modify values of the attributes for a nested relation at any level of nesting. Condition statements are used to locate the tuple in the relation where the values are to be changed. If no condition is specified, the atomic values of all tuples in the relation or relation-valued attribute will be changed to the new value. The code generator *codegen* creates the *query.e* file which contains the code to implement the *modify* method.

To illustrate, using the *employees* schema as defined in Appendix C, suppose that the employee David is working on a project named "BNAD" that must be renamed "TROY".

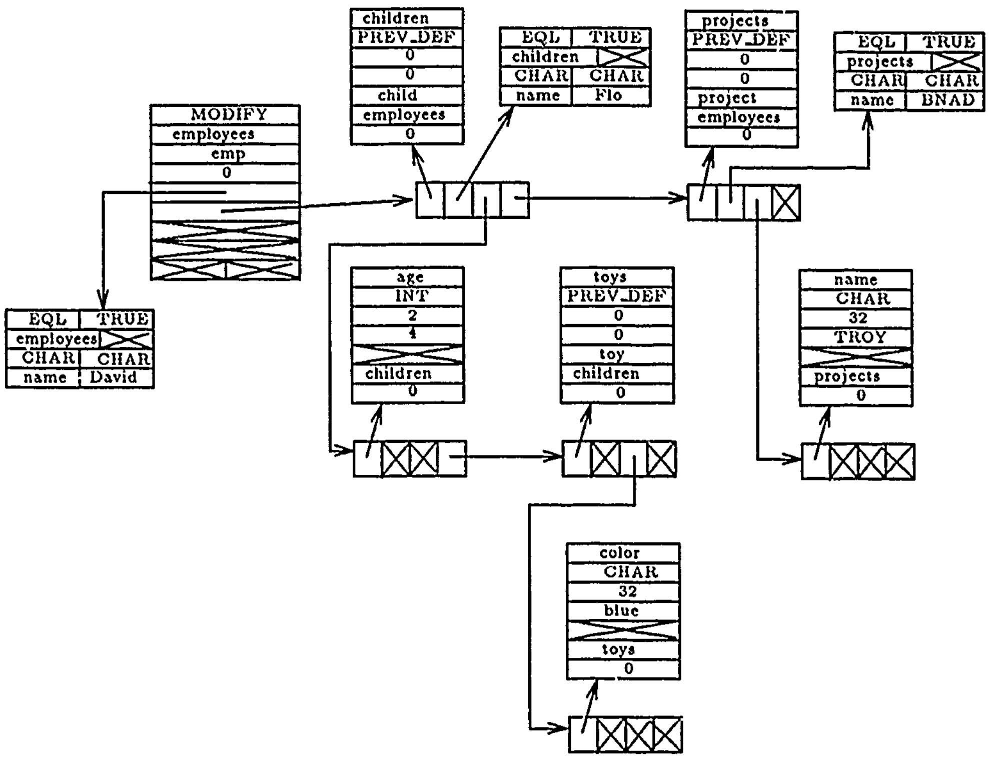

Figure 3.9. Example Plan Tree For *Modify* Query

Also assume we wish to change the age of his daughter, Flo, to four, and all her toys to blue. Figure 3.9 gives the plan tree representation of this query. The leaves of the tree are new values for attributes in the relation. The three attributes being changed (project name, child age, and toy color) are represented by three leaf nodes. The conditions represented by the predicate trees specify whether the change is occurring at selected tuples or to all tuples of the relation or relation-valued attribute.

The contents of *query.e* for the plan tree depicted in Figure 3.9 is given in Appendix H. The method scans *employees* until David is reached. Then David's children are scanned until Flo is reached. At this point, Flo's age is changed to four, and all her toys' colors are changed to blue. If we wanted to change the color of only some of Flo's toys, a predicate tree would appear on the *toys* list node specifying the select condition. Finally, David's projects are scanned until "BNAD" is found, which is renamed to "TROY".

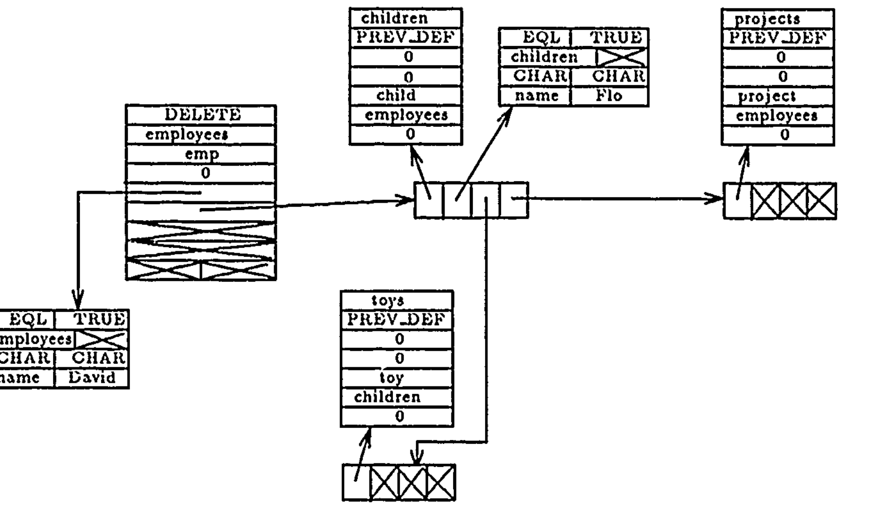

Figure 3.10. Example Plan Tree For *Delete* Query

3.3.5 Delete Method. The *delete* method can delete tuples from a nested relation at any level of nesting. Condition statements are used to locate the level(s) where the deletion is to occur. The code generator *codegen* creates the *query.e* file which contains the code to implement the *delete* method.

To illustrate, using the *employees* relation schema as defined in Appendix C, suppose all of David's projects and all of his daughter Flo's toys are to be deleted. Figure 3.10 gives the plan tree representation of this query. Only relation-valued attributes appear in the attribute list. A *list* node with a condition and a sublist indicates that the deletion does not occur at this level, but at some sub-level. If a *list* node has a condition but no sublist, only those tuples that meet the selection condition are deleted. If the *list* node has no condition, then all its tuples are deleted. When a tuple is deleted that has a relation-valued attribute in its schema, all tuples in its relation-valued attribute must be deleted.

Appendix I gives the contents of *query.e* for the plan tree of Figure 3.10. The method scans *employees* until David is reached. Then all of David's children are scanned until Flo is reached, and all her toys are deleted. David's projects are then deleted.

3.4 Code Generator

The code generator *codegen* generates a unique *query.e* file for each query by walking the plan tree.[1] The main program of *codegen* makes a function call to *traverse_plan_tree* to walk the plan tree and generate the code. After this is e, the main program calls *generate_main* which generates the main program of *query.e*.

3.4.1 Rationale Behind Design Decisions for the Code Generator. The end purpose for the Triton system is to be the backend for non-standard application tools. Application programs make calls to the database via embedded SQL/NF statements. When it came time to design how the Triton system would perform queries, two approaches were possible. The query implementor could either be (1) an *interpreter* that manipulates the database directly, or (2) a *code generator* that generates the code to implement the query.

There are advantages and disadvantages to both approaches. The interpreter approach would work well in an *ad hoc* query environment, since manipulation of the database is performed directly by the interpreter. The code generator would require substantial time to compile and run each generated query. The code generator approach works best when specific queries are known ahead of time, so that the code for those queries is already generated and compiled. The interpreter approach would take longer in this type of environment, since the query must be analyzed and executed on the fly. Because the intended environment for the Triton system does not require an *ad hoc* query capability, the code generator approach was chosen.

3.4.2 System Catalogs. The code generator uses two system catalogs (which are stored as relations) to record information about the relations in the database. These two catalogs, developed by Mankus (21) are called *Rel_table* and *Sym_table.* Figure 3.11 gives an example of the contents of *Rel_table* to hold the two relations, *employees* and *products*, which were used in the description of the *loops_join* method in Section 3.3.2. Figure 3.12 shows the corresponding contents of *Sym_table.*

[1]However, *codegen* does not create a *query.e* file for the create relation and delete relation access methods.

relIndex	*relName*	*relType*
0	employees	10
1	products	20

Figure 3.11. Contents of *Rel_table*

Rel_table has three attributes, *relIndex*, *relName*, and *relType*. The name of the relation is recorded in the *relName* attribute. The *relIndex* attribute serves as an index for *Rel_table*, and the *relType* attribute records the relation's schema and is a number corresponding to the index of *Sym_table* where the schema is defined.

Sym_table has seven attributes called *index*, *name*, *level*, *attr*, *numb*, *parent*, and *nest_table*. The *name* attribute holds the name of the schema or attribute. Schema names must be unique. The *level* attribute tells whether this entry is a schema or an attribute. The *attr* attribute describes the type of the attribute. Relation-valued attributes have the value PREV_DEFINED for *attr*, while schemas have ON_THE_FLY. The *numb* attribute records the number of characters if the type is CHAR, the number of bytes if the type is INT or FLOAT, or the number of attributes if the type is PREV_DEFINED or ON_THE_FLY. If the value for *level* is ATTRIBUTE, the *parent* attribute holds the index number of the parent schema. If the value for *attr* is PREV_DEFINED, the *nest_table* attribute holds the index number of its defining schema.

3.4.3 Operation of the Code Generator. All the methods except for the *loops_join* method is represented by a single node plan tree. However, the *loops_join* method is represented by a multi-node plan tree which is traversed by *codegen* in postorder. *Codegen* uses "statement templates" in the E programming language to create the *query.e* files (described in the previous section) corresponding to the plan tree. Recursion is used whenever a sublist is encountered in the plan tree, which greatly simplifies the actions of *codegen* and allows any level of nesting in the plan tree. Since *codegen* directly performs the actions necessary to create and delete relations, these are described in more detail below.

index	*name*	*level*	*attr*	*numb*	*parent*	*nest_table*
0	toy	SCHEME	ON_THE_FLY	2	-1	-2
1	color	ATTRIBUTE	CHAR	32	0	-2
2	name	ATTRIBUTE	CHAR	32	0	-2
3	child	SCHEME	ON_THE_FLY	3	-1	-2
4	name	ATTRIBUTE	CHAR	32	3	-2
5	age	ATTRIBUTE	INT	2	3	-2
6	toys	ATTRIBUTE	PREV_DEFINED	2	3	0
7	project	SCHEME	ON_THE_FLY	2	-1	-2
8	name	ATTRIBUTE	CHAR	32	7	-2
9	number	ATTRIBUTE	INT	2	7	-2
10	emp	SCHEME	ON_THE_FLY	5	-1	-2
11	name	ATTRIBUTE	CHAR	32	10	-2
12	age	ATTRIBUTE	INT	2	10	-2
13	dno	ATTRIBUTE	INT	2	10	-2
14	children	ATTRIBUTE	PREV_DEFINED	3	10	3
15	projects	ATTRIBUTE	PREV_DEFINED	2	10	7
16	manufacturer	SCHEME	ON_THE_FLY	3	-1	-2
17	location	ATTRIBUTE	CHAR	32	16	-2
18	name	ATTRIBUTE	CHAR	32	16	-2
19	phone	ATTRIBUTE	INT	2	16	-2
20	product	SCHEME	ON_THE_FLY	3	-1	-2
21	name	ATTRIBUTE	CHAR	32	20	-2
22	price	ATTRIBUTE	FLOAT	4	20	-2
23	manufacturers	ATTRIBUTE	PREV_DEFINED	3	20	16

Figure 3.12. Contents of *Sym_table*

3.4.3.1 Operation of Code Generator for Create_Rel. When *codegen* encounters a *plan* node where the method is "CREATE_REL", it creates the files describing the relation definitions. As an example, Appendix C shows the files that describe the relation definitions for the *employees* relation. In addition, *codegen* creates a batch file called *compile_schemas* which, when executed, will compile the relation definitions.

3.4.3.2 Operation of Code Generator for Drop_Rel. When *codegen* encounters a *plan* node where the method is "DROP_REL", it checks both *Rel_table* and *Sym_table* to see if the relation scheme is a defining scheme for another relation or if it is used within another scheme definition. If it is not being used, *codegen* deletes the contents of the relation, removes the corresponding scheme definition files, and deletes the scheme from the symbol table. After all appropriate schemas have been deleted, *codegen* deletes the relation from the relation table.

3.5 Summary

This chapter described how nested relations are represented in Triton using the constructs of the E programming language. Following this description, each of the methods implemented in this research effort were discussed. Finally, the chapter concluded with a look at the operation of Triton's code generator. Using the access and operator methods presented in this chapter, the following chapter evaluates the performance of the nested relational model versus the relational model and discusses the advantages and disadvantages of using the EXODUS toolkit in the development of the Triton system.

IV. *Analysis and Evaluation*

4.1 Overview

As stated in Chapter 1, the Triton system is intended to be the backend storage component for non-standard database applications, such as CAD, CASE, or OIS tools. The first application for the Triton system is the representation of a particular CASE methodology, the USAF IDEF$_0$ Structured Analysis language (22). Morris (23) defined a nested relational model and a relational model representation for IDEF$_0$. As a result of his analysis, he speculated that the speed of query execution for the NRM representation would be faster than that of the relational representation. The goal of this chapter is to assess Morris' speculation by implementing both NRM and relational representations in the Triton system, and comparing query performance in terms of query generation and execution times. We found that in both areas, the nested representation outperformed the relational version in terms of speed, particularly in code generation time. Following this comparison, we discuss the advantages and disadvantages of using the EXODUS toolkit for the development of the Triton system.

4.2 Comparison of the NRM and Relational Representations of the IDEF$_0$ Language Data

4.2.1 Schema Definitions. Using SQL/NF CREATE TABLE commands, the schema definitions for the relational and nested relational representations of the IDEF$_0$ Language Data are given in Appendices J and K, respectively. At the outermost level, only one SQL/NF CREATE TABLE command is needed for the NRM representation since all IDEF$_0$ language data is contained in one table. The data used to fill these tables is the same data used by Morris (23).

Because the interface between the SQL/NF front end and the backend of the Triton system is not yet developed, the SQL/NF CREATE TABLE commands were not used to generate the E language structures (the .e and .h files) that describe the NRM and relational representations of the IDEF$_0$ language data. Instead, the system catalogs were filled with data describing the schema of the two representations, and the E language structures of both representations were automatically generated using the *create_rel* method described in Chapter 3. Even without the use of the SQL/NF front end, generation of the E language structures was fairly straightforward.

4.2.2 Queries Used in this Comparison. Morris identified seven actions the CASE application tool might request from the database. These are:

1. Create all database tables
2. Load all database tables
3. Erase all database tables
4. Extract all data in database
5. Extract all drawing data for a particular sheet (diagram)
6. Extract data dictionary information for a particular activity
7. Extract data dictionary information for a particular data element

The last three activities (queries) in this list were selected for the comparison presented in this chapter, since the majority of the application program's queries will probably be one of these three.

Morris (23) presented the SQL/NF descriptions of each of these queries. According to his analysis, the relational representation requires eight subqueries to extract all drawing data for a particular sheet, whereas only one query is needed to extract the same information from the NRM database. In the same fashion, six subqueries are required in the relational version as compared to one query in the nested version to extract activity data dictionary information. To extract data element data dictionary information, twelve subqueries are required in the relational version and one query is needed in the nested version.

4.2.3 Method of Comparison. The two representations were compared with respect to code generation time and query execution time. Each query was generated and executed twenty times, and the minimum, maximum, variance, and average were calculated for each of these twenty runs. All reported times are in seconds, and all runs were accomplished on a Sun 3 workstation at approximately the same level of workload.

Table 4.1. Code Generation Times for Relational Drawing Data Query

Subquery #	Min Time	Max Time	Variance	Average Time
1	3.06	3.22	0.002080	3.092
2	2.36	2.50	0.000779	2.390
3	3.00	3.18	0.002378	3.021
4	3.00	3.12	0.001873	3.021
5	3.00	3.14	0.000938	3.017
6	2.36	2.50	0.000825	2.384
7	3.00	3.16	0.001831	3.029
8	4.28	4.42	0.001401	4.307
			Total	24.261

Table 4.2. Code Generation Times for Relational Activity Data Dictionary Query

Subquery #	Min Time	Max Time	Variance	Average Time
1	2.44	2.58	0.002521	2.465
2	2.34	2.48	0.000922	2.362
3	2.54	2.88	0.006496	2.573
4	3.02	3.16	0.002152	3.054
5	3.02	3.14	0.001301	3.038
6	3.78	3.94	0.002294	3.819
			Total	17.311

4.2.4 Comparison of Code Generation Times. Table 4.1 gives the code generation figures for the relational version of the drawing query. These are the figures obtained by running *codegen* to generate the code to implement the drawing query. Note that eight subqueries are needed, so the total gives the average total query generation time for the entire query. In the same fashion, Tables 4.2 and 4.3 give the code generation figures for the relational versions of the activity and data element data dictionary queries, respectively. For the nested version, since no subqueries are needed to extract the drawing data, activity data dictionary information and data element data dictionary information, each of these three retrievals involve only one query. Table 4.4 gives the code generation figures for all three queries. The low variance validates the fact that workload is about the same across the board for all runs.

Table 4.5 compares the code generation times for the relational version against the

Table 4.3. Code Generation Times for Relational Data Element Data Dictionary Query

Subquery #	Min Time	Max Time	Variance	Average Time
1	2.38	2.54	0.001906	2.413
2	2.32	2.54	0.004762	2.366
3	2.34	2.50	0.001057	2.374
4	2.36	2.58	0.003667	2.386
5	2.98	3.10	0.000733	2.992
6	2.84	3.00	0.003225	2.876
7	2.84	2.98	0.001806	2.858
8	2.34	2.46	0.000711	2.355
9	2.34	2.46	0.007368	2.350
10	2.94	3.16	0.002706	3.033
11	2.92	3.08	0.002754	2.968
12	3.02	3.16	0.000972	3.034
			Total	32.005

Table 4.4. Code Generation Times for Nested Queries

Query	Min Time	Max Time	Variance	Average Time
Drawing Data	7.40	7.66	0.006349	7.475
Activ Data Dict	3.10	3.26	0.001704	3.131
Data Elem Data Dict	6.06	6.22	0.002748	6.103

nested version for each of the three queries. Note that for each query, the generation time for the nested version takes less than one third the time for the equivalent query in the relational version. The difference in code generation times can be attributed to two factors: one, the relational version requires several subqueries to extract needed information and two, the increased complexity of the plan trees for the relational versions of the queries adversely impacts the speed of the code generator. Each of these factors is examined, in turn, below.

The fact that there are many subqueries needed in the relational version means that many functions of the code generator must be duplicated for each subquery, such as opening the output file to hold the generated query, checking to see if the main program should be generated, and, if it is, generation of the main program. These actions may seem

Table 4.5. Comparison of Code Generation Times

Query	Relational Generation Time	Nested Generation Time
Drawing Data	24.261	7.475
Activ Data Dict	17.311	3.131
Data Elem Data Dict	32.005	6.103

trivial, but can increase the run time of the code generator for queries made up of many subqueries.

Perhaps the most significant contribution to the code generation time is the complexity of the query's plan tree. In the relational version, there are 40 tables that make up the database, as compared to only one table in the nested version. For both the relational and nested versions, each table involved in a query must have at least one filescan node in the plan tree. Thus, queries in the relational version are represented as multi-node plan trees, with filescans for each table in the query and join nodes connecting them. However, the plan trees in the nested version of the queries contain only one filescan node, since all information is selected and projected from one table. The code generator sets up a separate iterator for each filescan and join node in the plan tree. Clearly, for multi-node plan trees, the job of *codegen* is much more involved. A good assumption seems to be that this complexity accounts for a large portion of the higher code generation times for the relational version as compared to the nested version.

The fact that the code generator always checks for the existence of sublist pointers in *list* nodes is a design characteristic that hinders the relational version, since all sublist pointers are null. However, elimination of this check would not make up the huge difference in code generation times.

4.2.5 Comparison of Query Execution Times. Table 4.6 presents the query exection figures for the relational version of the drawing query. Note that eight subqueries are needed, so the total gives the average total time to execute the entire query. In the same fashion, Tables 4.7 and 4.8 present the query execution figures for the relational versions of the activity and data element data dictionary queries, respectively. Table 4.9 presents the

Table 4.6. Execution Times for Relational Drawing Data Query

Subquery #	Min Time	Max Time	Variance	Average Time
1	0.04	0.30	0.003583	0.094
2	0.06	0.40	0.004901	0.172
3	0.06	0.30	0.003954	0.122
4	0.06	0.18	0.001536	0.091
5	0.04	0.28	0.003491	0.098
6	0.04	0.18	0.002454	0.083
7	0.04	0.38	0.006732	0.165
8	0.04	0.24	0.002173	0.084
			Total	0.660

Table 4.7. Execution Times for Relational Activity Data Dictionary Query

Subquery #	Min Time	Max Time	Variance	Average Time
1	0.02	0.10	0.000500	0.045
2	0.02	0.04	0.000067	0.024
3	0.04	0.14	0.000854	0.073
4	0.06	0.54	0.009663	0.170
5	0.08	0.32	0.003074	0.140
6	0.06	0.22	0.002754	0.108
			Total	0.702

NRM version query execution figures for the drawing query, activity data dictionary query, and the data element data dictionary query. Again, the low variance is a good indication that the workload was the same across the board for all runs.

Table 4.10 compares the query execution times for the relational version against the nested version for each of the three queries. As with code generation, the nested query exection time is less than the relational query execution time. The difference in query execution time can be attributed to two factors: one, the large number of joins in the relational version, and two, increased disk access time due to a lack of clustering in the relational version.

As stated earlier, all information is contained in one table in the nested version, whereas the relational version breaks the $IDEF_0$ language data into 40 normalized tables.

Table 4.8. Execution Times for Relational Data Element Data Dictionary Query

Subquery #	Min Time	Max Time	Variance	Average Time
1	0.06	0.20	0.001915	0.121
2	0.04	0.22	0.002504	0.071
3	0.04	0.18	0.001637	0.075
4	0.06	0.26	0.002580	0.103
5	0.04	0.16	0.001746	0.081
6	0.06	0.24	0.002942	0.095
7	0.06	0.18	0.001637	0.095
8	0.04	0.16	0.001853	0.080
9	0.04	0.14	0.001246	0.084
10	0.08	0.34	0.003515	0.119
11	0.12	0.32	0.002699	0.194
12	0.14	0.24	0.000500	0.145
			Total	1.263

Table 4.9. Query Execution Times for Nested Queries

Query	Min Time	Max Time	Variance	Average Time
Drawing Data	0.48	1.00	0.024399	0.551
Activ Data Dict	0.52	0.86	0.004037	0.695
Data Elem Data Dict	0.74	0.92	0.001367	0.769

Each time data is correlated between two or more tables, a join is executed. While no joins are required in the nested version (since all data is contained in one table), some of the relational subqueries require as many as four joins. As stated in Chapter 3, the join method implemented is a nested loops join method, which is not very efficient. For each tuple of one relation, each tuple of the second relation is checked to see if the join criteria are met. A more efficient algorithm such as a sort-merge join or a hash join would probably speed up the execution time of the relational version. However, even with the speedup realized with a more efficient join algorithm, the execution time of the relational version would probably still exceed that of the nested relational version, since no joins are required in the nested relational version. The difference in execution times is not as great as expected, probably due to the limited number of tuples in the relational tables. As the

Table 4.10. Comparison of Query Execution Times

Query	Relational Execution Time	Nested Execution Time
Drawing Data	0.660	0.551
Activ Data Dict	0.702	0.695
Data Elem Data Dict	1.263	0.769

size of the database grows, the execution times should increasingly favor the nested version of the database.

The second reason for the faster execution of the nested queries is that disk access times are longer for relational queries since tables are not automatically clustered on disk. The EXODUS storage manager automatically clusters tuples of a collection on disk to decrease access time. Of course, this means that attributes in the tuples are clustered, including relation-valued attributes. Since the nested version has only one overall table (with several nested tables), all data in the nested version should theoretically be clustered on disk. However, in the relational version, no explicit clustering is specified. The EXODUS storage manager allows "near" hints as to how to cluster data, but none are specified for the relational version. If these are used, the execution time for the relational version might realize some speedup.

4.2.6 Summary of Comparison. The goal of this comparison was to demonstrate the empirical advantage of the NRM over the relational model for the storage and retrieval of complex data. This advantage was demonstrated by comparing the two models on code generation time and query execution time. In both areas, the nested version outperformed the relational version, particularly in code generation time. However, between generation and execution time, execution time is most important. This is because queries will be generated once, but will be executed many times. However, the performance of both activities must be considered, since both play a crucial role in determining the processing speed of the Triton system.

4.3 Evaluation of the EXODUS Toolkit in this Research Effort

Implementation of a DBMS is not a trivial undertaking, but the use of extensible systems has eased some of the burden of this task. The use of the EXODUS extensible toolkit reduced the development time of the Triton nested relational database system. The following subsections present the advantages and disadvantages of using the EXODUS toolkit in the development of the Triton system by specifically evaluating the following:

- collections
- classes
- persistence
- data representation
- optimizer generator
- storage manager

4.3.1 Collections. The Triton system is built on the nested relational data model, allowing relation-valued attributes in relations. The nested relational data model is mapped very nicely using the *collection* generator class supplied by EXODUS in the E programming language. Relation-valued attributes are represented using collections of collections. Unfortunately, EXODUS only provides the capability for sequential scanning of collections, making access via a search key slow for large relations. The only way around this shortcoming is to build indexes on every frequently accessed or sufficiently large relation and nested relation.

The E programming language does not permit subclasses to be derived from the *collection* class, which is a limitation. Since a collection is an *unordered* group of objects, assume we want to derive a subclass of the *collection* class called *ordered_collection*, which is an *ordered* group of objects. E does not allow this derivation from the *collection* class.

A good performance aspect of the *collection* class is that items in a collection are clustered on disk, including subcollections. This automatic clustering reduces access time and frees the implementor of having to worry about the grouping of data on disk.

4.3.2 Classes. The use of classes is an excellent object-oriented technique for hiding the actual representation of data and allowing access to subobjects through predefined constructor and member functions. The use of generator classes should prove very useful in the development of special indexing techniques for the Triton system. This is because a generic index (such as B^+-tree) can be developed using a generator class which takes the type as a parameterized input. A specific index is then instantiated for a particular type using the generator class specification.

4.3.3 Persistence. In E, persistent objects are declared explicitly by using the keyword **persistent** before the declaration of the object. However, only **db** types can be declared persistent, since the use of a **db** type specifies that the object is a disk object as opposed to a main memory object. Unfortunately, every subcomponent of a **dbclass** must be a **db** type, which can cause some problems if the database implementor chooses to change a main memory class to a persistent class; not only does the class have to be changed to a **dbclass**, but all subobjects in the class must be changed to their **db** counterparts.

4.3.4 Data Representation. As stated in Section 4.3.1, nested relations are conveniently mapped using the E *collection* class by mapping them directly into the underlying programming language. This allows the relation definitions (the *.e and *.h files) to be compiled into object files using the E compiler. Compiled queries are linked with the object files containing the relation definitions and executed to perform the query. The advantage of this method is that queries are executed quickly, since they are in machine code. However, the long compilation and linking time makes this method prohibitive for dealing with ad hoc queries.

4.3.5 Optimizer Generator. At the present time, Triton's optimizer component has not been developed. However, the intention is to use the EXODUS optimizer generator to generate this component. The EXODUS optimizer generator takes as input (1) a set of operators, (2) a set of methods that implement the operators, (3) transformation rules that describe equivalence-preserving transformations of query trees, and (4) implementation rules that describe how to replace an operator with a specific method. Using these rules, a specific optimizer is generated for the particular application.

We chose to use the EXODUS optimizer generator for Triton because the relational algebra used by the system lends itself to EXODUS' rule-based method. This modular approach to database development will reduce the amount of code required for implementation of the Triton system. The only unique code Triton's developers will need to write will be the additional functions that are called by the optimizer when implementing a specific operator or access method. For example, take the following implementation rule:

```
project (loops_join (1,2)) by loops_join (1,2) combine_ljp;
```

This means that if there is a *loops_join* followed by a projection, it can become a special case of *loops_join* that will perform the join and the projection in the same step. To do this, the function *combine_ljp* is called to combine the projection list and the argument list of the *loops_join*. The function *combine_ljp* is an example of the code that must be written by the Triton system developers. However, this is far less involved than developing an optimizer from scratch. With the use of accurate cost functions, the generated optimizer should perform as well as a custom-built one.

4.3.6 Storage Manager. The use of the EXODUS storage manager greatly reduced the development time of the Triton system by freeing the developer from worrying about Triton's storage component. The storage manager's procedural interface allows access to database objects without having to know how they are actually being manipulated. As an example, persistent objects are automatically mapped to permanent storage locations through the use of the **persistent** keyword. Data clustering is performed by using the *collection* class and is transparent to the developer. Access to collections of objects is provided by the E programming language via built in procedure calls to the storage manager, such as *scan*, *in ... new*, and *delete*.

4.3.7 Overall Evaluation of the EXODUS Toolkit. EXODUS was very helpful in the development of the Triton system. In particular, the E programming language elegantly maps nested relations using the *collection* class. While EXODUS only supplies sequential access to collections, implementation of indices should be simple using generator classes. The storage manager handles persistence and automatically clusters nested relations on disk. In addition, the storage manager's procedural interface made interaction with the storage manager virtually transparent. Use of the EXODUS optimizer generator should greatly reduce the time it takes to create Triton's optimizer.

4.4 Summary

The first half of this chapter demonstrated the viability of the nested model by comparing the performance of the nested and relational representations of $IDEF_0$ language data based on code generation time and query exection time. In both areas, the nested version performed faster. The second half of this chapter discussed the advantages and disadvantages of using the EXODUS toolkit in the developement of the Triton nested relational database system.

V. Conclusions and Recommendations

5.1 Overview

This chapter summarizes and draws conclusions about the work presented in this thesis. Recommendations for further work as it pertains to the Triton nested database system is also discussed.

5.2 Summary of Research

The goal of the research presented in this thesis was to develop a backend to the Triton nested database system to manipulate the data in the database as specified by some query. The goal was not to provide a plethora of access methods, but to give Triton the ability to process queries of any type (except for data restructuring queries) and to lay the foundation for future work in this area.

The work accomplished in this research effort may be summarized as follows:

1. Design and implementation of the operator methods of the Colby algebra to handle multiple levels of nesting, including:
    - The *filescan* method that implements a project and/or a select
    - The *loops_join* method that implements a join as well as a project and/or select
2. Design and implementation of the access methods to modify data in the database, including:
    - The *store_values* method that adds data to a relation at any level or levels of nesting
    - The *modify* method that modifies data in a relation at any level or levels of nesting
    - The *delete* method that deletes data from a relation at any level or levels of nesting
    - The *create_rel* method that adds a new relation to the database

- The *drop_rel* method that deletes a relation from the database

3. Design and implementation of the code generator, *codegen*, to implement the operator and access methods
4. Testing of methods and code generator
5. Comparison of the performance of the nested relational model versus its normalized (1NF) version using the operator methods and code generator:
   - Development of an E representation of a sample set of relational $IDEF_0$ language data (23)
   - Development of an E representation of nested relational $IDEF_0$ language data (23)
   - Design of programs to load data into both representations of the $IDEF_0$ language data
   - Creation of queries for both representations to evaluate performance of the NRM against the traditional relational model
   - Comparison of the NRM against the relational model based on code generation time and query execution time
6. Discussion of the advantages and disadvantages of using the EXODUS toolkit in this research effort

5.3 Conclusions

By using Triton to store a nested and relational representation of CASE data and using the methods implemented in this research to query that data, we found that the nested relational model outperforms the relational model in terms of code generation time and query execution time. These results demonstrate the viability of the nested relational model for the representation of complex data. In addition, the EXODUS toolkit provided several necessary tools that greatly reduced the development time of the Triton system. However, the use of the EXODUS toolkit did constrain this development, and future efforts must attempt to overcome these constraints. With this in mind, the following section recommends possible areas for improvement in the Triton system.

5.4 Recommendations

As stated earlier, the purpose for research presented in this thesis was to give Triton the ability to process queries of any type (except for data restructuring queries) and to lay the foundation for future work in this area. Actions must be taken to bring the Triton query processing system up to full capability. These actions include:

- Design of the *nest* and *unnest* operator methods to implement the data restructuring actions of the Colby algebra
- Generation of the optimizer to properly optimize the query tree and call appropriate methods

Additional actions need to be taken to give Triton more than just a cursory set of operator and access methods. This includes:

- Design of additional operator methods, such as *merge_join*, or *index_join* to more efficiently handle the join operator
- Development of indexing techniques and access methods to use them:
  - I recommend looking at Bertino and Kim's work in this area (3), but especially recommend implementing a similar indexing strategy as the one used by the ANDA system (11)

As a performance issue, sequential access to the system catalogs is inefficient. I suggest either establishing indices into the catalogs, or implementing schema information as persistent structures in main memory, as is done in the Ariel system (14). This would speed up query execution substantially, especially as the size of the system catalogs grows.

I recommend that future work address the possibility of making a query interpreter in addition to the code generator. This would make testing easier on application developers. The interpreter would have to be linked with the schema definition files for all relations in the database. If such an effort is undertaken, the developer should consider implementing the methods as a class hierarchy of plan objects as is done in Ariel (14).

Appendix A. *Colby Relational Algebra*

The use of the nested relational model requires the development of operators to manipulate nested data. Latha S. Colby (9) devised a recursive algebra for nested relations that builds on the traditional relational algebra operators. Her operators allow the retrieval of information from any level of nesting in a relation without first "flattening out" the relation to fit the standard relational model. To permit nesting, Colby employed set operators and redefined the select (σ), project (π) and join ($\bowtie$) operators of the relational model and introduced two new operators, nest (ν) and unnest (μ). For the purposes of this exposition, I assume that the select, project and join operators are understood as they apply to non-nested relations.[1] Their use by Colby is defined below.

A.1 Select (σ).

The select operator is used to extract tuples from the relation that satisfy a specific selection criteria. Its structure is defined as

$$\sigma\ (\text{relation}_{condition}\ (\textit{select list}))$$

The *select list* is used to indentify conditions on relation-valued attributes (RVAs). The *select list* can be null or has the following recursive form:

$$(\text{RVA}_{condition}\ (\textit{select list}))$$

For example, using the database schema provided in Figure 1.3, a query to find all the employees who are older than 35 that have male children is written as

$$\sigma\ (\text{employee}_{emp\_age>35}\ (\text{children}_{sex=M}))$$

which yields the relation shown in Figure A.1.

[1]For a definition of the relational algebra operators for non-nested relations, see (8).

dept	*emp_name*	*emp_age*	*emp_ssn*	*children*		
				child_name	*child_age*	*sex*
Adv	T. Therrien	43	555-12-3434	John	13	M
				Matthew	11	M
Pers	C. Dunlap	37	624-35-8152	George	5	M

Figure A.1. Employees Over 35 With Male Children

emp_name	*child_name*
John Smith	Jeramie
	Todd
Michael Taylor	Susan
Tina Therrien	Laura
	John
	Matthew
Carla Dunlap	George
	Janis

Figure A.2. Project Out Employee and Children Names

A.2 Project (π).

The project operator is used to extract specific attributes of a relation. Its structure is defined as

$$\pi \ ((\textit{project list}) \ \text{relation})$$

where a *project list* indentifies the list of attributes to be extracted from the relation. If the *project list* contains any relation-valued attributes, those attributes may have their own *project list*. For example, a qu ry to retrieve all employees' names and their childrens' names from the *employee* relation s written as

$$\pi \ ((\text{emp\_name, children (child\_name)}) \ \text{employee})$$

which yields the relation shown in Figure A.2.

name	*classes*	
	course	*grade*
Jeramie	Math	B
	Science	B^+
Cheryl	Chemistry	B^+
	Comp Sci	A^-
	Math	B
Laura	Poli Sci	C
John	Biology	B^+
	English	B^-
Mike	Physics	A
	Math	A
Matthew	Chemistry	B^+
	Science	B
	English	A^-

Figure A.3. The *Student* Relation

A.3 Join (⋈).

The join operator joins a relation to another relation or to any relation valued attribute (RVA) of a relation. Its structure is as follows:

$$\bowtie_{\theta} \text{ (relation}_1 \text{ (}\textit{join path}\text{), relation}_2\text{)}$$

The condition is specified in the θ portion of the query. If the *join path* is null, the two relations will be joined at the highest level. However, the *join path* may identify the level of relation$_1$ where the join is to take place. The *join path* is defined as:

$$\text{(RVA of relation}_1 \text{ (}\textit{join path} \text{ of RVA of relation}_1\text{))}$$

For example, suppose there is a separate *student* relation as shown in Figure A.3 in addition to the nested *employee* relation. The query to join the children attribute of the employee relation to the student relation where children name is equal to the student name is

$$\bowtie_{child\_name=name} \text{ (employee (children), student)}$$

which yields the relatio· shown in Figure A.4.

dept	*emp_name*	*emp_age*	*emp_ssn*	*children*				
				child_name	*child_age*	*sex*	*classes*	
							course	*grade*
Mktg	Smith	27	237-46-3567	Jeramie	8	M	Math	B
							Science	B^+
Adv	Therrien	43	555-12-3434	Laura	18	F	Poli Sci	C
				John	13	M	Biology	B^+
							English	B^-
				Matthew	11	M	Chemistry	B^+
							Science	B
							English	A^-

Figure A.4. The *Employee* Relation Joined (on Children) to the *Student* Relation

dept	*emp_name*	*emp_age*	*emp_ssn*	*child_name*	*child_age*	*sex*
Mktg	J. Smith	27	237-46-3567	Jeramie	8	M
Mktg	J. Smith	27	237-46-3567	Todd	4	M
Rsrch	M. Taylor	31	395-73-8901	Susan	3	F
Adv	T. Therrien	43	555-12-3434	Laura	18	F
Adv	T. Therrien	43	555-12-3434	John	13	M
Adv	T. Therrien	43	555-12-3434	Matthew	11	M
Pers	C. Dunlap	37	624-35-8152	George	5	M
Pers	C. Dunlap	37	624-35-8152	Janis	3	F

Figure A.5. The Flat *Employee* Relation

A.4 Nest (ν).

The nest operator is a restructuring operator that groups certain specified attributes of a relation into a nested version of the relation. Its structure is

$$\nu \text{ attribute list} \rightarrow \text{RVA name (relation)}$$

where the attribute list specifies the attributes to be nested under the title of the RVA name.

For example, a query to take the flattened version of the *employee* relation as shown in Figure A.5 and nest *child_name, child_age* and *sex* into a relation valued attribute called *children* would be written as

$$\nu \text{ child\_name, child\_age, sex} \rightarrow \text{children (employee)}$$

dept	*emp_name*	*emp_age*	*emp_ssn*	*children*		
				child_name	*child_age*	*sex*
Mktg	J. Smith	27	237-46-3567	Jeramie	8	M
				Todd	4	M
Rsrch	M. Taylor	31	395-73-8901	Susan	3	F
Adv	T. Therrien	43	555-12-3434	Laura	18	F
				John	13	M
				Matthew	11	M
Pers	C. Dunlap	37	624-35-8152	George	5	M
				Janis	3	F

Figure A.6. The Nested *Employee* Relation

which creates the nested version of the *employee* relation as shown in Figure A.6.

A.5 Unnest (μ).

The unnest operator is the inverse operation of the nest operator. It flattens out a relation by duplicating the atomic attributes of a tuple for each occurance of the relation valued attributes. Its structure is as follows:

μ relation-valued attribute (relation)

To illustrate its operation, a query to take the nested version of the *employee* relation as shown in Figure A.6 and flatten it would be written as:

μ children (employee)

creating the unnested or flattened version of the *employee* relation as shown in Figure A.5.

Appendix B. *Definition of Data Structures*

The design and implementation of the Triton nested database system utilizes several data structures first developed by Mankus (21) that I modified for my use. Because these structures play such a vital role in understanding the design of my operator and access methods, each is explained in detail.

B.1 Plan Node

The output of the query optimizer is a pointer to the plan tree that structurally represents the user's request. Figure B.1 shows the structure of a *plan* node.

The fields of the *plan* node are used as follows:

- method – specifies the operator or access method to be used (for example *filescan, loops_join, create_rel*, or *store_values*, to name a few)
- argument.name – the name of the input relation (used only if the method is FILESCAN)
- argument.reltype – the type of the input relation (used only if the method is FILESCAN)
- argument.struct_num – an integer field that records the greatest structure number of the temporary relation templates that hold the intermediate data for answering the query
- argument.pred – a pointer to a predicate tree of *pred* nodes which represents the conditions on the highest level of the nested relation
- argument.list – a linked list of *list* nodes which identify attributes
- argument.aux_pred – an additional place to hang a condition (used only in the LOOPS_JOIN method when a selection is simultaneously taking place)
- argument.aux_list – an additional place to hang a linked list of *list* nodes (used only in the LOOPS_JOIN method when a projection is simultaneously taking place)
- input[0] and input[1] – pointers to *plan* nodes

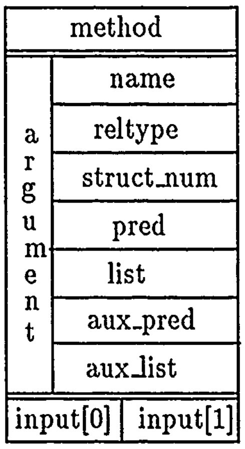

Figure B.1. A *Plan* Node

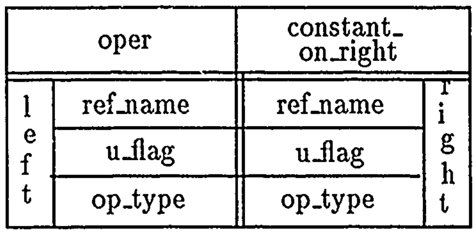

Figure B.2. A *Pred* Node

B.2 Pred Node

The *pred* and *aux_pred* portions of the plan node can each point to a tree of *pred* nodes that represent query conditions. Figure B.2 shows the structure of a *pred* node.

The fields of the *pred* node are used as follows:

- oper – identifies the operator which can be AND, OR, NOT, LES (less than), GRT (greater than), LEQ (less than or equal to), GEQ (greater than or equal to), EQL (equal to), and NEQ (not equal to)
- constant_on_right – boolean variable to identify whether the value stored in the right operand is a constant or a variable
- ref_name – records the relation name that the attribute identified in the operand portion pertains to

- uflag – a union flag to record whether the value stored in the op_type field is a character pointer, integer, float or a pointer to another pred node
- op_type – holds either a character pointer, integer, float, or a pointer to another pred node

The *pred* node has a left and right operand, which are inputs to the operator. All operators are binary except for the NOT operator, which is unary and only uses the left operand. If the operator is an AND, OR, or NOT, the op_type holds a pointer to another *pred* node; otherwise, it will hold either a pointer to a character, an integer value, or a float value. The left operand holds the name of the attribute for the condition, while the right operand holds the value of a constant or may hold the name of another attribute in the relation. If the latter case is true, the value of the *constant_on_right* field would be false.

To demonstrate the structure of a predicate tree, the predicate

emp_age>40 and dept="Marketing"

is structurally represented in Figure B.3. In this figure, two *pred* nodes are used to represent the two conditions, while a third *pred* node connects these conditions with an AND. Since the two attributes *emp_age* and *dept* are being compared to a constant value, the value of *constant_on_right* for both of the nodes at the bottom of the tree is true. The left operand is always the name of an attribute, thus the left *op_type* field will always hold a character value, which is the name of the attribute.

B.3 List Node

The *list* and *aux_list* portions of the *plan* node can each point to a linked list of *list* nodes which represent attribute lists for use in performing projections. Figure B.4 shows the structure of a *list* node.

The fields of the *list* node are used as follows:

- attr – points to an *attrdesc* node which holds a description of the attribute and is defined in detail below

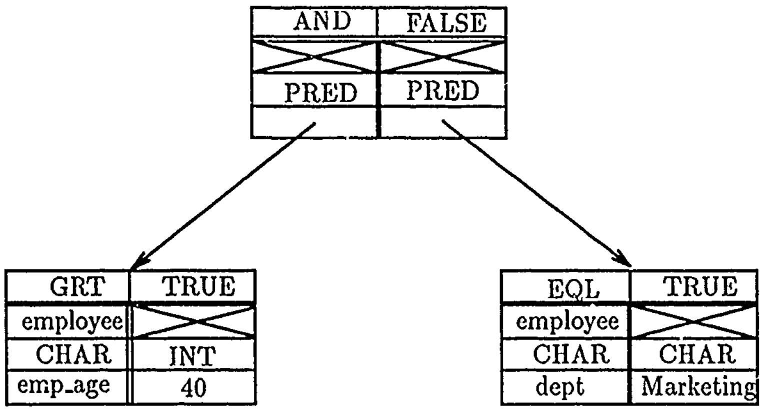

Figure B.3. A Predicate Tree

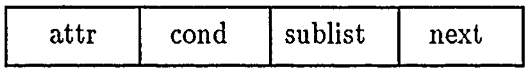

Figure B.4. A *List* Node

- cond – points to a predicate tree of *pred* nodes (only used if the attribute is relation-valued and there is a selection condition that applies to the attributes of the relation-valued attribute)
- sublist – points to a sublist of *list* nodes (only used if the attribute is relation-valued and only some of the attributes of the relation-valued attribute are being projected out)
- next – points to the next *list* node in the attribute list

As stated above, the *list* structure represents an attribute list. Relation-valued attributes may have a condition (which points to a predicate tree of *pred* nodes) and a sublist (which points to the attributes of the relation-valued attribute that are to be projected out). If the attribute is relation-valued and the sublist is null, all attributes of the relation-valued attribute are to be projected out. Through the use of sublists, projections can be represented at any level of nesting in the relation. The list structure can be used to record a selection, projection, or both a selection and a projection.

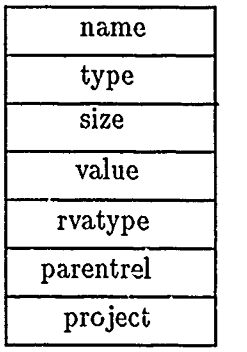

Figure B.5. An *Attrdesc* Node

B.4 Attrdesc Node

The *attr* field of the *list* node points to an *attrdesc* node which holds information about the attribute. Figure B.5 shows the structure of an *attrdesc* node.

The fields of the *attrdesc* node are used as follows:

- name – the name of the attribute
- type – the type of the attribute (CHAR if the attribute is a character pointer, INT if the attribute is an integer, FLOAT if the attribute is a float, or PREV_DEFINED if the attribute is relation-valued)
- size – the number of characters if the type is CHAR, the number of bytes if the type is INT or FLOAT (if the type is PREV_DEFINED, then size is 0)
- value – holds a character string if the type is CHAR, an integer if the type is INT, a float if the type is FLOAT (used only to specify the value of an attribute if a new tuple is being added to the relation or if the value of the attribute is being changed)
- rvatype – holds the name of the type of tuple to be stored in the relation-valued attribute (only used if type is PREV_DEFINED)
- parentrel – holds the name of the relation to which this attribute belongs
- project – an integer value that holds the temporary relation number (used when a temporary relation template is made to hold the intermediate results of a projection or a join)

Appendix C. *Data for the Employees Relation*

The following files specify the schema shown in Figure C.1 for the *employees* nested relation. The *employees* relation is a collection of tuples of type *emp*. *Children* and *projects* are relation-valued attributes of *emp*, and *toys* is a relation-valued attribute of *children*. The *employees* relation is used in the body of this thesis to demonstrate the function of the operator and access methods.

The schema for the nested *employees* relation is contained in a series of files. There is a *.e* and a *.h* file describing the *emp*, *child*, *toy*, and *project* schemas. There is also a *.e* and *.h* file describing the overall *employees* schema. The *.h* file contains the structure of the relation, while the *.e* file contains the code to implement the constructor and member functions. Separating the schemas into pairs of files was necessary to ensure the schemas are not multiply defined if they are used more than once in a relation. In the code that follows, only the *.e* file for the *emp* relation is given. In the interest of space and because the implementation of the constructor and member functions for the *child*, *toy*, and *project* relations are nearly identical, *child.e*, *toy.e*, and *project.e* are not given here.

			children				*projects*	
name	*age*	*dno*	*toys*		*age*	*name*	*name*	*number*
			color	*name*				

Figure C.1. Schema for the *Employees* Nested Relation

```
/*************************************************/
/*  The following is the content of employee.h   */
/*************************************************/
#ifndef EMPLOYEES_H
#ifndef EMP_H
#include "emp.h"
#endif

  dbclass empRVA:collection[emp];

#define EMPLOYEES_H
#endif
```

```
/***********************************************/
/*   The following is the content of employee.e   */
/***********************************************/
#include <stream.h>
#include <stdio.h>

#ifndef EMPLOYEES_H
#include "employees.h"
#endif

persistent empRVA employees;

/***********************************************/
/*     The following is the content of emp.h      */
/***********************************************/
#ifndef EMP_H
#ifndef CHILD_H
#include "child.h"
#endif

#ifndef PROJECT_H
#include "project.h"
#endif

dbstruct emp {
  dbchar name[32];
  dbint  age;
  dbint  dno;
  dbclass childRVA:collection[child];
  childRVA children;
  dbclass projectRVA:collection[project];
  projectRVA projects;
public:
  emp (char *, int, int);
  char * get_name();
  void change_name (char *);
  int get_age();
  void change_age (int);
  int get_dno();
  void change_dno (int);
  void print (emp *);
};

#define EMP_H
#endif
```

```
/*************************************************/
/*     The following is the content of emp.e     */
/*************************************************/
#include <stream.h>
#include <stdio.h>

#ifndef EMP_H
#include "emp.h"
#endif

emp::emp (char * nameAtom, int ageAtom, int dnoAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
  age = ageAtom;
  dno = dnoAtom;
}

char * emp::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

void emp::change_name (char * nameAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
}

int emp::get_age() {
  return (age);
}

void emp::change_age (int ageAtom) {
  age = ageAtom;
}

int emp::get_dno() {
  return (dno);
}
```

```
void emp::change_dno (int dnoAtom) {
  dno = dnoAtom;
}

void emp::print (emp * emp_ptr) {
  cout << form ("emp_name: %s\n", emp_ptr->get_name());
  cout << form ("emp_age: %d\n", emp_ptr->get_age());
  cout << form ("emp_dno: %d\n", emp_ptr->get_dno());
  emp & emp_ref_ptr = * emp_ptr;
  iterate (child * child_ptr = emp_ref_ptr.children.scan())
    child_ptr -> print (child_ptr);
  iterate (project * project_ptr = emp_ref_ptr.projects.scan())
    project_ptr -> print (project_ptr);
}

/************************************************/
/*   The following is the content of child.h    */
/************************************************/
#ifndef CHILD_H
#ifndef TOY_H
#include "toy.h"
#endif

dbstruct child {
  dbclass toyRVA:collection[toy];
  toyRVA toys;
  dbint  age;
  dbchar name[32];
public:
  child (int, char *);
  int get_age();
  void change_age (int);
  char * get_name();
  void change_name (char *);
  void print (child *);
};
```

```
#define CHILD_H
#endif
/**************************************************/
/*     The following is the content of toy.h      */
/**************************************************/
#ifndef TOY_H
dbstruct toy {
  dbchar color[32];
  dbchar name[32];
public:
  toy (char *, char *);
  char * get_color();
  void change_color (char *);
  char * get_name();
  void change_name (char *);
  void print (toy *);
};


/**************************************************/
/*   The following is the content of project.h    */
/******  ******************************************/
#ifndef PROJECT_H
dbstruct project {
  dbchar name[32];
  dbint  number;
public:
  project (char *, int);
  char * get_name();
  void change_name (char *);
  int get_number();
  void change_number (int);
  void print (project *);
};


#define PROJECT_H
#endif
```

Appendix D. *Data for the Products Relation*

The following files specify the schema shown in Figure D.1 for the *products* nested relation. The *products* relation is a collection of tuples of type *product*. *Manufacturers* is a relation-valued attribute of *product*. The *products* relation is used in the body of this thesis to demonstrate the function of the *loops_join* operator method.

The schema for the nested *products* relation is contained in a series of files. There is an *.e* and an *.h* file describing the *product* and *manufacturer* schemas. There is also an *.e* and *.h* file describing the overall *products* schema. The *.h* file contains the structure of the relation, while the *.e* file contains the code to implement the constructor and member functions. Separating the schemas into pairs of files was necessary to ensure the schemas are not multiply defined if they are used more than once in a relation. In the code that follows, only the *.e* file for the *product* relation is given. In the interest of space and because the implementation of the constructor and member functions for *manufacturer* are nearly identical, the content of *manufacturer.e* is not given here.

name	*price*	*manufacturers*		
		location	*name*	*phone*

Figure D.1. Schema for the *Products* Nested Relation

```
/*************************************************/
/*   The following is the content of products.h   */
/*************************************************/
#ifndef PRODUCTS_H
#ifndef PRODUCT_H
#include "product.h"
#endif

  dbclass productRVA:collection[product];

#define PRODUCTS_H
#endif
```

```
/***********************************************/
/*   The following is the content of products.e   */
/***********************************************/
#include <stream.h>
#include <stdio.h>

#ifndef PRODUCTS_H
#include "products.h"
#endif

persistent productRVA products;

/***********************************************/
/*   The following is the content of product.h    */
/***********************************************/
#ifndef PRODUCT_H
#ifndef MANUFACTURER_H
#include "manufacturer.h"
#endif

dbstruct product {
  dbchar name[32];
  dbfloat price;
  dbclass manufacturerRVA:collection[manufacturer];
  manufacturerRVA manufacturers;
public:
  product (char *, float);
  char * get_name();
  void change_name (char *);
  float get_price();
  void change_price (float);
  void print (product *);
};

#define PRODUCT_H
#endif
```

```
/**************************************************/
/*   The following is the content of product.e    */
/**************************************************/
#include <stream.h>
#include <stdio.h>

#ifndef PRODUCT_H
#include "product.h"
#endif

product::product (char * nameAtom, float priceAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
  price = priceAtom;
}

char * product::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

void product::change_name (char * nameAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
}

float product::get_price() {
  return (price);
}

void product::change_price (float priceAtom) {
  price = priceAtom;
}
```

```
void product::print (product * product_ptr) {
  cout << form ("product_name: %s\n", product_ptr->get_name());
  cout << form ("product_price: %f\n", product_ptr->get_price());
  product & product_ref_ptr = * product_ptr;
  iterate (manufacturer * manufacturer_ptr = product_ref_ptr.
                              manufacturers.scan())
    manufacturer_ptr -> print (manufacturer_ptr);
}

/*******************************************************/
/*   The following is the content of manufacturer.h    */
/*******************************************************/
#ifndef MANUFACTURER_H
dbstruct manufacturer {
  dbchar location[32];
  dbchar name[32];
  dbint  phone;
public:
  manufacturer (char *, char *, int);
  char * get_location();
  void change_location (char *);
  char * get_name();
  void change_name (char *);
  int get_phone();
  void change_phone (int);
  void print (manufacturer *);
};


#define MANUFACTURER_H
#endif
```

Appendix E. *Filescan Method*

The following code implements the *filescan* query depicted in Figure 3.6 and described in Section 3.3.1. In the Triton system, the code in this appendix would be written to a file called *query.e* by the E code generator *codegen* to implement the query. The comments that appear in the code in this appendix are not generated by *codegen* but have been put in by the author to explain the operation of the *filescan*.

```
#include <stream.h>
#include <stdio.h>

#ifndef EMPLOYEES_H
#include "employees.h"
#endif

extern persistent empRVA employees;

/*******************************************************/
/*  Temporary relation template -- holds the projected */
/*  attributes of the toys relation.                   */
/*******************************************************/

dbstruct temp1 {
  dbchar name[32];
public:
  temp1 (char *);
  char * get_name();
};

/****************************************/
/*  Constructor function code for temp1  */
/****************************************/

temp1::temp1 (char * nameAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
}
```

```
/**************************************************/
/*  Code for implementing member function of temp1  */
/**************************************************/

char * temp1::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

/******************************************************/
/*  Temporary relation template -- holds the projected  */
/*  attributes of the children relation.                */
/******************************************************/

dbstruct temp2 {
  dbchar name[32];
  dbint  age;
  dbclass temp1RVA:collection[temp1];
  temp1RVA temp1_rels;
public:
  temp2 (char *, int);
  char * get_name();
  int get_age();
};

/****************************************/
/*  Constructor function code for temp2  */
/****************************************/

temp2::temp2 (char * nameAtom, int ageAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
  age = ageAtom;
}
```

```
/****************************************************/
/*  Code for implementing member functions of temp2  */
/****************************************************/

char * temp2::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

int temp2::get_age() {
  return (age);
}

/*******************************************************/
/*  Temporary relation template -- holds the projected  */
/*  attributes of the employees relation.               */
/*******************************************************/

dbstruct temp3 {
  dbchar name[32];
  dbint  age;
  dbclass temp2RVA:collection[temp2];
  temp2RVA temp2_rels;
public:
  temp3 (char *, int);
  char * get_name();
  int get_age();
};

/*****************************************/
/*  Constructor function code for temp3  */
/*****************************************/

temp3::temp3 (char * nameAtom, int ageAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
  age = ageAtom;
}
```

```
/***************************************************/
/*  Code for implementing member functions of temp3  */
/***************************************************/

char * temp3::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

int temp3::get_age() {
  return (age);
}

/****************************************************************/
/*                      FILESCAN ITERATOR                       */
/*                                                              */
/*  This iterator will perform the filescan of the employees    */
/*  relation and will yield a pointer to the resultant projected */
/*  tuple.  (i.e. a pointer to a tuple of type temp3)            */
/****************************************************************/

iterator temp3 * filescan_temp3()
    {
    /******************************/
    /*  scan the employee relation  */
    /******************************/
    iterate (emp * left_tuple_ptr1 = employees.scan())
        {
        /************************/
        /*  selection condition  */
        /************************/
        if (left_tuple_ptr1->get_age() > 30)
            {
            /*****************************************************/
            /*  copy the projected atomic attributes from employees  */
            /*  into temp3                                           */
            /*****************************************************/
            temp3 * t3 = new temp3 (left_tuple_ptr1->get_name(),
                                    left_tuple_ptr1->get_age());
```

```
/*********************************************************/
/*  set up a reference pointer to descend one level of  */
/*  nesting in temp3                                     */
/*********************************************************/
temp3 & temp3_ref = * t3;
/*********************************************************/
/*  set up a reference pointer to descend one level of  */
/*  nesting in employees                                 */
/*********************************************************/
emp & left_source_ref1 = * left_tuple_ptr1;
/********************************/
/*  scan the children relation  */
/********************************/
iterate (child * left_tuple_ptr2 = left_source_ref1.
                                    children.scan())
    {
    /**************************/
    /*  selection condition  */
    /**************************/
    if (left_tuple_ptr2->get_age() < 5)
        {
        /***********************************************/
        /*  copy the projected atomic attributes from  */
        /*  children into temp2                        */
        /***********************************************/
        temp2 * t2 = in (temp3_ref.temp2_rels) new temp2
               (left_tuple_ptr2->get_name(),
                left_tuple_ptr2->get_age());
        /***********************************************/
        /*  set up a reference pointer to descend one  */
        /*  level of nesting in temp2                  */
        /***********************************************/
        temp2 & temp2_ref = * t2;
        /***********************************************/
        /*  set up a reference pointer to descend one  */
        /*  level of nesting in children               */
        /***********************************************/
        child & left_source_ref2 = * left_tuple_ptr2;
```

```
                        /****************************/
                        /*  scan the toys relation  */
                        /****************************/
                        iterate (toy * left_tuple_ptr3 = left_source_ref2.
                                                       toys.scan())
                            {
                            /*****************************************/
                            /*  copy the projected atomic attributes */
                            /*  from toys into temp1                 */
                            /*****************************************/
                            temp1 * t1 = in (temp2_ref.temp1_rels) new temp1
                                              (left_tuple_ptr3->get_name());
                            }
                        }
                    }
                /***********************************************************/
                /* yield the projected tuple, which is a pointer to a temp3 */
                /***********************************************************/
                yield (t3);
                }
            }
        }
/**********************************************************************/
/*                            MAIN PROGRAM                            */
/*  The main program will call the filescan iterator and print out the */
/*  contents of the projected and selected relation.                   */
/**********************************************************************/
main()
    {
    iterate (temp3 * left_tuple_ptr1 = filescan_temp3())
        {
        cout << form ("employees_name: %s\n",
                          left_tuple_ptr1 -> get_name());
        cout << form ("employees_age: %d\n",
                          left_tuple_ptr1 -> get_age());
        temp3 & left_source_ref1 = * left_tuple_ptr1;
        iterate (temp2 * left_tuple_ptr2 = left_source_ref1.
                          temp2_rels.scan())
            {
            cout << form ("children_name: %s\n",
                          left_tuple_ptr2 -> get_name());
            cout << form ("children_age: %d\n",
                          left_tuple_ptr2 -> get_age());
            temp2 & left_source_ref2 = * left_tuple_ptr2;
            iterate (temp1 * left_tuple_ptr3 = left_source_ref2.
                  temp1_rels.scan())
           {
           cout << form ("toys_name: %s\n",
                  left_tuple_ptr3 -> get_name());
           }
        }
     }
  }
```

Appendix F. *Loops_Join Method*

The following code implements the *loops_join* query depicted in Figure 3.7 and described in Section 3.3.2. In the Triton system, the code in this appendix would be written to a file called *query.e* by the E code generator *codegen* to implement the query. The comments that appear in the code in this appendix are not generated by *codegen* but have been put in by the author to explain the operation of the *loops_join*.

```
#include <stream.h>
#include <stdio.h>

#ifndef EMPLOYEES_H
#include "employees.h"
#endif

extern persistent empRVA employees;

/*******************************************************/
/*  Temporary relation template -- holds the projected  */
/*  attributes of the toys relation.                    */
/*******************************************************/

dbstruct temp1 {
  dbchar name[32];
public:
  temp1 (char *);
  char * get_name();
};

/****************************************/
/*  Constructor function code for temp1  */
/****************************************/

temp1::temp1 (char * nameAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
}
```

```
/****************************************************/
/*  Code for implementing member function of temp1  */
/****************************************************/

char * temp1::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

/*********************************************************/
/*  Temporary relation template -- holds the projected  */
/*  attributes of the employees relation.                */
/*********************************************************/

dbstruct temp2 {
  dbchar name[32];
  dbclass temp1RVA:collection[temp1];
  temp1RVA temp1_rels;
public:
  temp2 (char *);
  char * get_name();
};

/******************************************/
/*  Constructor function code for temp2  */
/******************************************/

temp2::temp2 (char * nameAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
}
```

```
/**************************************************/
/*  Code for implementing member function of temp2  */
/**************************************************/

char * temp2::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

/*******************************************************************/
/*                         FILESCAN ITERATOR                        */
/*                                                                  */
/*  This iterator will perform the filescan of the employees        */
/*  relation and will yield a pointer to the resultant projected   */
/*  tuple.  (i.e. a pointer to a tuple of type temp2)               */
/*******************************************************************/

iterator temp2 * filescan_temp2()
    {
    /********************************/
    /*  scan the employee relation  */
    /********************************/
    iterate (emp * left_tuple_ptr1 = employees.scan())
        {
        /*******************************************************/
        /*  copy the projected atomic attribute from employees  */
        /*  into temp2                                          */
        /*******************************************************/
        temp2 * t2 = new temp2 (left_tuple_ptr1->get_name());
        /*******************************************************/
        /*  set up a reference pointer to descend one level of  */
        /*  nesting in temp2                                    */
        /*******************************************************/
        temp2 & temp2_ref = * t2;
        /*******************************************************/
        /*  set up a reference pointer to descend one level of  */
        /*  nesting in employees                                */
        /*******************************************************/
        emp & left_source_ref1 = * left_tuple_ptr1;
    /*******************************/
    /*  scan the children relation  */
    /*******************************/
    iterate (child * left_tuple_ptr2 = left_source_ref1.children.scan())
      {
      /*****************************************************/
      /*  set up a reference pointer to descend one level of  */
      /*  nesting in children                                 */
      /*****************************************************/
      child & left_source_ref2 = * left_tuple_ptr2;
      /***************************/
      /*  scan the toys relation  */
      /***************************/
      iterate (toy * left_tuple_ptr3 = left_source_ref2.toys.scan())
        {
        /***********************************************/
        /*  copy the projected atomic attribute from toys  */
        /*  into temp1                                     */
        /***********************************************/
        temp1 * t1 = in (temp2_ref.temp1_rels) new temp1
                       (left_tuple_ptr3-[illegible]me());
        }
      }
    /*********************************************************/
    /*  yield the projected tuple, which is a pointer to a temp2  */
    /*********************************************************/
    yield (t2);
    }
  }

#ifndef PRODUCTS_H
#include "products.h"
#endif

extern persistent productRVA products;
```

```
/*********************************************************/
/*  Temporary relation template -- holds the projected  */
/*  attributes of the products relation.                */
/*********************************************************/

dbstruct temp3 {
  dbchar name[32];
  dbclass manufacturerRVA:collection[manufacturer];
  manufacturerRVA manufacturers;
public:
  temp3 (char *);
  char * get_name();
};

/******************************************/
/*  Constructor function code for temp3  */
/******************************************/

temp3::temp3 (char * nameAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
}

/****************************************************/
/*  Code for implementing member function of temp3  */
/****************************************************/

char * temp3::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}
```

```
/********************************************************************/
/*                      FILESCAN ITERATOR                           */
/*                                                                  */
/*  This iterator will perform the filescan of the products         */
/*  relation and will yield a pointer to the resultant projected    */
/*  tuple.  (i.e. a pointer to a tuple of type temp3)               */
/********************************************************************/

iterator temp3 * filescan_temp3()
    {
    /********************************/
    /*  scan the products relation  */
    /********************************/
    iterate (product * left_tuple_ptr1 = products.scan())
        {
        /********************************************************/
        /*  copy the projected atomic attributes from products  */
        /*  into temp3                                          */
        /********************************************************/
        temp3 * t3 = new temp3 (left_tuple_ptr1->get_name());
        /*******************************************************/
        /*  set up a reference pointer to descend one level of */
        /*  nesting in temp3                                   */
        /*******************************************************/
        temp3 & temp3_ref = * t3;
        /*******************************************************/
        /*  set up a reference pointer to descend one level of */
        /*  nesting in products                                */
        /*******************************************************/
        product & left_source_ref1 = * left_tuple_ptr1;
        /*************************************/
        /*  scan the manufacturers relation  */
        /*************************************/
        iterate (manufacturer * left_tuple_ptr2 = left_source_ref1.
                                  manufacturers.scan())
            {
            /**********************************************************/
            /* copy the projected atomic attributes from manufacturers */
            /* into the manufacturer relation-valued attribute of temp3 */
            /**********************************************************/
            manufacturer * manufacturer_ptr = in (temp3_ref.manufacturers)
                   new manufacturer (left_tuple_ptr2->get_location(),
                   left_tuple_ptr2->get_name(),
                   left_tuple_ptr2->get_phone());
            }
        /*************************************************************/
        /*  yield the projected tuple, which is a pointer to a temp3  */
        /*************************************************************/
        yield (t3);
        }
    }

/*******************************************************/
/*  Temporary relation template -- holds the attributes  */
/*  of the join for the manufacturers relation-valued    */
/*  attribute of the join                                */
/*******************************************************/

dbstruct temp4 {
  dbchar location[32];
  dbchar name[32];
  dbint  phone;
public:
  temp4 (char *, char *, int);
  char * get_location();
  char * get_name();
  int get_phone();
};

/*****************************************/
/*  Constructor function code for temp4  */
/*****************************************/

temp4::temp4 (char * locationAtom, char * nameAtom, int phoneAtom) {
  dbchar * dest;
  dest = location;
  while (*dest++ = *locationAtom++);
  dest = name;
  while (*dest++ = *nameAtom++);
  phone = phoneAtom;
}
```

```
/*****************************************************/
/*  Code for implementing member functions of temp4  */
/*****************************************************/

char * temp4::get_location() {
  dbchar * source = location;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

char * temp4::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

int temp4::get_phone() {
  return (phone);
}

/*********************************************************/
/*  Temporary relation template -- holds the attributes  */
/*  of the join at the join_level                         */
/*********************************************************/

dbstruct temp5 {
  dbchar toys_name[32];
  dbchar products_name[32];
  dbclass temp4RVA:collection[temp4];
  temp4RVA temp4_rels;
public:
  temp5 (char *, char *);
  char * get_toys_name();
  char * get_products_name();
};
```

```
/****************************************/
/*  Constructor function code for temp5  */
/****************************************/

temp5::temp5 (char * toys_nameAtom, char * products_nameAtom) {
  dbchar * dest;
  dest = toys_name;
  while (*dest++ = *toys_nameAtom++);
  dest = products_name;
  while (*dest++ = *products_nameAtom++);
}

/***************************************************/
/*  Code for implementing member functions of temp5  */
/***************************************************/

char * temp5::get_toys_name() {
  dbchar * source = toys_name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

char * temp5::get_products_name() {
  dbchar * source = products_name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

/*******************************************************/
/*  Temporary relation template -- holds the attributes  */
/*  of the join for the employees attribute of the join  */
/*******************************************************/

dbstruct temp6 {
  dbchar name[32];
  dbclass temp5RVA:collection[temp5];
  temp5RVA temp5_rels;
public:
  temp6 (char *);
  char * get_name();
};
```

```
/****************************************/
/*  Constructor function code for temp6  */
/****************************************/

temp6::temp6 (char * nameAtom) {
  dbchar * dest;
  dest = name;
  while (*dest++ = *nameAtom++);
}

/**************************************************/
/*  Code for implementing member function of temp6  */
/**************************************************/

char * temp6::get_name() {
  dbchar * source = name;
  char * dest = new char[32];
  char * start = dest;
  while (*dest++ = *source++);
  return (start);
}

/*****************************************************************/
/*                      LOOPS_JOIN ITERATOR                      */
/*                                                               */
/*  This iterator will perform the join of the products relation */
/*  to the employees relation and will yield a pointer to the    */
/*  resultant tuple.  (i.e. a pointer to a tuple of type temp6)  */
/*****************************************************************/

iterator temp6 * loops_join_temp6()
    {
    /****************************************************/
    /*  scan the temp2 relation using filescan iterator  */
    /****************************************************/
    iterate (temp2 * left_tuple_ptr1 = filescan_temp2())
        {
        /******************************************************/
        /*  copy the atomic attributes from left relation into  */
        /*  temp6                                               */
        /******************************************************/
        temp6 * t6 = new temp6 (left_tuple_ptr1->get_name());
```

```
/*******************************************************/
/*  set up a reference pointer to descend one level of  */
/*  nesting in temp6                                    */
/*******************************************************/
temp6 & temp6_ref = * t6;
/*******************************************************/
/*  set up a reference pointer to descend one level of  */
/*  nesting in the left relation                        */
/*******************************************************/
temp2 & left_source_ref1 = * left_tuple_ptr1;
/*****************************/
/*  scan the temp1 relation  */
/*****************************/
iterate (temp1 * left_tuple_ptr2 = left_source_ref1.
                                   temp1_rels.scan())
    {
    /***************************************************/
    /*  scan the temp3 relation using filescan iterator  */
    /***************************************************/
    iterate (temp3 * right_tuple_ptr1 = filescan_temp3())
        {
        /********************/
        /*  join condition  */
        /********************/
        if (strcmp (left_tuple_ptr2->get_name(), right_tuple_ptr1->
                                get_name()) == 0)
            {
            /***************************************************/
            /*  copy the atomic attributes from left and right  */
            /*  relations into temp5                            */
            /***************************************************/
            temp5 * t5 = in (temp6_ref.temp5_rels) new temp5
                         (left_tuple_ptr2->get_name(),
                          right_tuple_ptr1->get_name());
            /**********************************************/
            /*  set up a reference pointer to descend one  */
            /*  level of nesting in temp5                  */
            /**********************************************/
            temp5 & temp5_ref = * t5;
            /**********************************************/
            /*  set up a reference pointer to descend one  */
            /*  level of nesting in right relation         */
            /**********************************************/
            temp3 & right_source_ref1 = * right_tuple_ptr1;
```

```
                    /*****************************************/
                    /*  scan temp3's manufacturers relation  */
                    /*****************************************/
                    iterate (manufacturer * right_tuple_ptr2 =
                            right_source_ref1.manufacturers.scan())
                        {
                        /*******************************************/
                        /*  copy the atomic attributes from right  */
                        /*  relation into temp4                    */
                        /*******************************************/
                        temp4 * t4 = in (temp5_ref.temp4_rels) new temp4
                                 (right_tuple_ptr2->get_location(),
                                  right_tuple_ptr2->get_name(),
                                  right_tuple_ptr2->get_phone());
                        }
                    }
                }
            }
        /*********************************************************/
        /*  yield the joined tuple, which is a pointer to a temp6  */
        /*********************************************************/
        yield (t6);
        }
    }

/*********************************************************************/
/*                             MAIN PROGRAM                          */
/*                                                                   */
/*  The main program will call the join iterator and print out the   */
/*  contents of the joined relation.                                 */
/*********************************************************************/
main()
    {
    iterate (temp6 * left_tuple_ptr1 = loops_join_temp6())
        {
        cout << form ("employees_name: %s\n",
                           left_tuple_ptr1 -> get_name());
        temp6 & left_source_ref1 = * left_tuple_ptr1;
        iterate (temp5 * left_tuple_ptr2 = left_source_ref1.
                           temp5_rels.scan())
            {
            cout << form ("toys_toys_name: %s\n", left_tuple_ptr2 ->
                                get_toys_name());
            cout << form ("products_products_name: %s\n", left_tuple_ptr2 ->
                                get_products_name());
            temp5 & left_source_ref2 = * left_tuple_ptr2;
            iterate (temp4 * left_tuple_ptr3 = left_source_ref2.
                                        temp4_rels.scan())
                {
                cout << form ("manufacturers_location: %s\n",
                               left_tuple_ptr3 -> get_location());
                cout << form ("manufacturers_name: %s\n",
                               left_tuple_ptr3 -> get_name());
                cout << form ("manufacturers_phone: %d\n",
                               left_tuple_ptr3 -> get_phone());
                }
            }
        }
    }
```

Appendix G. *Store_Values Method*

The following code implements the *store_values* query depicted in Figure 3.8 and described in Section 3.3.3. In the Triton system, the code would be written to a file called *query.e* by the E code generator *codegen* to implement the query. The comments that appear in the code in this appendix are not generated by *codegen* but have been put in by the author to explain the operation of the *store_values* method.

```
#include <stream.h>
#include <stdio.h>

#ifndef EMPLOYEES_H
#include "employees.h"
#endif

extern persistent empRVA employees;

main()
    {
    /********************************/
    /*  scan the employee relation  */
    /********************************/
    iterate (emp * left_tuple_ptr1 = employees.scan())
        {
        /*************************/
        /*  condition statement  */
        /*************************/
        if (strcmp (left_tuple_ptr1->get_name(),"David") == 0)
            {
            /********************************************************/
            /*  set up a reference pointer to descend one level of  */
            /*  nesting in employees                                */
            /********************************************************/
            emp & left_source_ref1 = * left_tuple_ptr1;
            /********************************/
            /*  scan the children relation  */
            /********************************/
            iterate (child * left_tuple_ptr2 = left_source_ref1.
                                               children.scan())
                {
```

```
            /*************************/
            /*  condition statement  */
            /*************************/
            if (strcmp (left_tuple_ptr2->get_name(),"Florence") == 0)
                {
                /********************************************************/
                /*  set up a reference pointer to descend one level of  */
                /*  nesting in children                                 */
                /********************************************************/
                child & left_source_ref2 = * left_tuple_ptr2;
                /******************************************/
                /* insert two new "toy" tuples into toys  */
                /******************************************/
                toy * toy_ref = in (left_source_ref2.toys)
                                 new toy ("car", "black");
                toy * toy_ref = in (left_source_ref2.toys)
                                 new toy ("truck", "blue");
                }
            }
        /************************************************/
        /*  insert a new "project" tuple into projects  */
        /************************************************/
        project * project_ref = in (left_source_ref1.projects)
                         new project ("AWANA", 384);
        }
    }
}
```

Appendix H. *Modify Method*

The following code implements the *modify* query depicted in Figure 3.9 and described in Section 3.3.4. In the Triton system, the code would be written to a file called *query.e* by the E code generator *codegen* to implement the query. The comments that appear in the code in this appendix are not generated by *codegen* but have been put in by the author to explain the operation of the *modify* method.

```
#include <stream.h>
#include <stdio.h>

#ifndef EMPLOYEES_H
#include "employees.h"
#endif

extern persistent empRVA employees;

main()
    {
    /********************************/
    /*  scan the employee relation  */
    /********************************/
    iterate (emp * left_tuple_ptr1 = employees.scan())
        {
        /************************/
        /*  condition statement  */
        /************************/
        if (strcmp (left_tuple_ptr1->get_name(),"David") == 0)
            {
            /****************************************************/
            /*  set up a reference pointer to descend one level of  */
            /*  nesting in employees                                */
            /****************************************************/
            emp & left_source_ref1 = * left_tuple_ptr1;
            /********************************/
            /*  scan the children relation  */
            /********************************/
            iterate (child * left_tuple_ptr2 = left_source_ref1.
                             children.scan())
                {
```

```
        /*************************/
        /*  condition statement  */
        /*************************/
        if (strcmp (left_tuple_ptr2->get_name(),"Flo") == 0)
            {
            /*************************************************/
            /*  invoke member function to change child's age  */
            /*************************************************/
            left_tuple_ptr2->change_age (4);
            /**********************************************/
            /*  set up a reference pointer to descend one  */
            /*  level of nesting in children               */
            /**********************************************/
            child & left_source_ref2 = * left_tuple_ptr2;
            /****************************/
            /*  scan the toys relation  */
            /****************************/
            iterate (toy * left_tuple_ptr3 = left_source_ref2.
                                               toys.scan())
                {
                /*************************************/
                /*  invoke member function to change  */
                /*  the color of the toys             */
                /*************************************/
                left_tuple_ptr3->change_color ("blue");
                }
            }
        }
    /********************************/
    /*  scan the projects relation  */
    /********************************/
    iterate (project * left_tuple_ptr4 = left_source_ref1.
                                           projects.scan())
        {
        /*************************/
        /*  condition statement  */
        /*************************/
```

```
           if (strcmp (left_tuple_ptr4->get_name(),"BNAD") == 0)
                {
                /***************************************/
                /*  invoke member function to change   */
                /*  the name of the project            */
                /***************************************/
                left_tuple_ptr4->change_name ("TROY");
                }
            }
        }
    }
}
```

Appendix I. *Delete Method*

The following code implements the *delete* query depicted in Figure 3.10 and described in Section 3.3.5. In the Triton system, this code is written to a file called *query.e* by the E code generator *codegen* to implement the query. The comments that appear in the code in this appendix are not generated by *codegen* but have been put in by the author to explain the operation of the *delete* method.

```
#include <stream.h>
#include <stdio.h>

#ifndef EMPLOYEES_H
#include "employees.h"
#endif

extern persistent empRVA employees;

main()
    {
    /********************************/
    /*  scan the employee relation  */
    /********************************/
    iterate (emp * left_tuple_ptr1 = employees.scan())
        {
        /************************/
        /*  condition statement  */
        /************************/
        if (strcmp (left_tuple_ptr1->get_name(),"David") == 0)
            {
            /******************************************************/
            /*  set up a reference pointer to descend one level of  */
            /*  nesting in employees                                */
            /******************************************************/
            emp & left_source_ref1 = * left_tuple_ptr1;
            /********************************/
            /*  scan the children relation  */
            /********************************/
            iterate (child * left_tuple_ptr2 = left_source_ref1.
                                                 children.scan())
                {
```

```
            /*************************/
            /*  condition statement  */
            /*************************/
            if (strcmp (left_tuple_ptr2->get_name(),"Flo") == 0)
                {
                /********************************************************/
                /*  set up a reference pointer to descend one level of  */
                /*  nesting in children                                  */
                /********************************************************/
                child & left_source_ref2 = * left_tuple_ptr2;
                /****************************/
                /*  scan the toys relation  */
                /****************************/
                iterate (toy * left_tuple_ptr3 = left_source_ref2.toys.scan())
                    {
                    /**************************/
                    /*  delete the toy tuple  */
                    /**************************/
                    delete left_tuple_ptr3;
                    }
                }
            }
        /********************************/
        /*  scan the projects relation  */
        /********************************/
        iterate (project * left_tuple_ptr4 = left_source_ref1.projects.scan())
            {
            /******************************/
            /*  delete the project tuple  */
            /******************************/
            delete left_tuple_ptr4;
            }
        }
    }
}
```

Appendix J. *SQL/NF Create Table Definition for the Relational Version of the IDEF$_0$ Language Data*

The SQL/NF statements to create the schema for the relational version of the IDEF$_0$ language data are as follows:

```
CREATE TABLE act2act_table (
             parent_node INT 4,
             child_node INT 4)

CREATE TABLE act2data_table (
             node_id INT 4,
             data_id INT 4,
             icom_type CHAR 2)

CREATE TABLE act2hist_table (
             node_id INT 4,
             hist_id INT 4)

CREATE TABLE act2ref_table (
             node_id INT 4,
             ref_id INT 4)

CREATE TABLE activity_table (
             node_id INT 4,
             node CHAR 21,
             name CHAR 26,
             project_id INT 4,
             author_id INT 2,
             version CHAR 11,
             date CHAR 9,
             x INT 2,
             y INT 2,
             visible_DRE INT 1,
             sheet_id INT 4)

CREATE TABLE act_changes_table (
             node_id INT 4,
             changes CHAR 61)

CREATE TABLE act_descr_table (
             node_id INT 4,
             line_no INT 2,
             desc_line CHAR 61)

CREATE TABLE alias_table (
             data_id INT 4,
             name CHAR 26,
             where_used CHAR 26,
             comment CHAR 26)
```

```
CREATE TABLE analyst_table (
             author_id INT 2,
             author CHAR 21)

CREATE TABLE arrow_table (
             symbol_id INT 4,
             arrow_type INT 1)

CREATE TABLE boundary_table (
             symbol_id INT 4,
             icom_code CHAR 3)

CREATE TABLE data2data_table (
             parent_data INT 4,
             child_data INT 4)

CREATE TABLE data2label_table (
             data_id INT 4,
             label_id INT 4)

CREATE TABLE data2ref_table (
             data_id INT 4,
             ref_id INT 4)

CREATE TABLE data2value_table (
             data_id INT 4,
             value_id INT 4)

CREATE TABLE data_changes_table (
             data_id INT 4,
             changes CHAR 61)

CREATE TABLE data_descr_table (
             data_id INT 4,
             line_no INT 2,
             desc_line CHAR 61)

CREATE TABLE data_elem_table (
             data_id INT 4,
             name CHAR 26,
             project_id INT 4,
             author_id INT 2,
             version CHAR 11,
             date CHAR 9)

CREATE TABLE data_range_table (
             data_id INT 4,
             range_data CHAR 61)

CREATE TABLE data_type_table (
             data_id INT 4,
             type CHAR 26)

CREATE TABLE data_value_table1 (
             value_id INT 4,
             value CHAR 16)

CREATE TABLE dot_table (
             symbol_id INT 4,
             dot_type INT 1)

CREATE TABLE footnote_table (
             graf_id INT 4,
             x INT 2,
             y INT 2)

CREATE TABLE feo_table (
             graf_id INT 4,
             picture CHAR 61)

CREATE TABLE graphics_table (
             graf_id INT 4,
             sheet_id INT 4)

CREATE TABLE hist_call_table (
             hist_id INT 4,
             hist_proj CHAR 21,
             hist_node CHAR 13)

CREATE TABLE label_table (
             label_id INT 4,
             name CHAR 11,
             x INT 2,
             y INT 2,
             sheet_id INT 4)

CREATE TABLE min_max_table (
             data_id INT 4,
             minimum CHAR 16,
             maximum CHAR 16)

CREATE TABLE note_table (
             graf_id INT 4,
             label INT 2,
             x INT 2,
             y INT 2)

CREATE TABLE note_text_table (
             graf_id INT 4,
             line_no INT 2,
             text_line CHAR 61)

CREATE TABLE project_table (
             project_id INT 4,
             name CHAR 13)

CREATE TABLE reference_table (
             ref_id INT 4,
             line_no INT 2,
             ref_line CHAR 61)
```

```
CREATE TABLE ref_type_table (
             ref_id INT 4,
             type_ref CHAR 26)

CREATE TABLE segment_table (
             seg_id INT 4,
             data_id INT 4,
             sheet_id INT 4,
             xs INT 2,
             ys INT 2,
             xe INT 2,
             ye INT 2)

CREATE TABLE sheet_table (
             sheet_id INT 4,
             c_number INT 4)

CREATE TABLE squiggle_table (
             graf_id INT 4,
             x1 INT 2,
             y1 INT 2,
             x2 INT 2,
             y2 INT 2,
             x3 INT 2,
             y3 INT 2,
             x4 INT 2,
             y4 INT 2)

CREATE TABLE tunnel_table (
             symbol_id INT 4,
             seg_id INT 4,
             sheet_id INT 4,
             x INT 2,
             y INT 2)

CREATE TABLE to_from_all_table (
             symbol_id INT 4,
             tfa_label CHAR 2)

CREATE TABLE symbol_table (
             symbol_id INT 4,
             tunnel_type INT 1)

CREATE TABLE turn_table (
             symbol_id INT 4,
             turn_type INT 1,
             node_id INT 4,
             hist_id INT 4)
```

Appendix K. *SQL/NF Create Table Definition for the Nested Version of the $IDEF_0$ Language Data*

The SQL/NF statement to create the schema for the nested version of the $IDEF_0$ language data is as follows:

```
CREATE TABLE projects (
            project_name CHAR 13,
            TABLE activities (
                  node_id INT 4,
                  node CHAR 21,
                  name CHAR 26,
                  author CHAR 21,
                  version CHAR 11,
                  date CHAR 9,
                  changes CHAR 61,
                  c_number INT 4,
                  parent CHAR 26,
                  TABLE act_descr (
                        line_no INT 2,
                        desc_line CHAR 61),
                  TABLE references (
                        ref_type CHAR 26,
                        TABLE ref_lines (
                              line_no INT 2,
                              line_ref CHAR 61)),
                  TABLE hist_calls (
                        hist_proj CHAR 13,
                        hist_node CHAR 21),
                  TABLE data_elems (
                        data_name CHAR 26,
                        icom_type CHAR 2),
                  TABLE children (
                        node_name CHAR 26),
            TABLE data_elements (
                  data_id INT 4,
                  name CHAR 26,
                  author CHAR 21,
                  version CHAR 11,
                  date CHAR 9,
                  changes CHAR 61,
                  parent CHAR 26,
                  TABLE data_descr (
                        line_no INT 2,
                        desc_line CHAR 61),
                  TABLE references (
                        ref_type CHAR 26,
                        TABLE ref_lines (
                              line_no INT 2,
                              line_ref CHAR 61)),
        TABLE aliases (
              name CHAR 26,
              where_used CHAR 26,
              comment CHAR 26),
        TABLE min_maxes (
              data_type CHAR 26,
              minimum CHAR 16,
              maximum CHAR 16),
        TABLE ranges (
              data_type CHAR 26,
              range_val CHAR 61),
        TABLE values (
              data_type CHAR 26,
              actual_value CHAR 16),
        TABLE activities (
              node_name CHAR 26,
              icom_type CHAR 2),
        TABLE children (
              data_name CHAR 26)),
  TABLE sheets (
        c_number INT 4,
        node CHAR 21,
        name CHAR 26,
        author CHAR 21,
        version CHAR 11,
        date CHAR 9,
        TABLE boxes (
              node CHAR 21,
              name CHAR 26,
              x INT 2
              y INT 2,
              visible_dre INT 2),
        TABLE segments (
              data_id INT 4,
              TABLE location (
                    xs INT 2,
                    ys INT 2,
                    xe INT 2,
                    ye INT 2),
              TABLE symbols (
                    x INT 2,
                    y INT 2,
                    symbol_type CHAR 26,
                    type_symbol CHAR 26)),
        TABLE squiggles (
              x1 INT 2,
              y1 INT 2,
              x2 INT 2,
              y2 INT 2,
              x3 INT 2,
              y3 INT 2,
              x4 INT 2,
              y4 INT 2),
        TABLE meta_notes (
              label CHAR 2.
              x INT 2,
        y INT 2,
        TABLE note_texts (
              line_no INT 2,
              text_line CHAR 61)),
  TABLE foot_notes (
        label INT 2,
        xm INT 2,
        ym INT 2,
        xn INT 2,
        yn INT 2,
        TABLE note_texts (
              line_no INT 2,
              text_line CHAR 61)),
  TABLE feos (
        label CHAR 2,
        x INT 2,
        y INT 2,
        picture CHAR 61),
  TABLE labels (
        data_id INT 4,
        name CHAR 11,
        x INT 2,
        y INT 2)))
```

Bibliography

1. Armstrong, James R. *Chip-Level Modeling with VHDL.* Englewood Cliffs, NJ: Prentice-Hall, 1989.
2. Batory, D. S., et al. "GENESIS: An Extensible Database Management System." In Zdonik, Stanley B. and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 500–518, San Mateo, CA: Morgan Kaufmann, 1990.
3. Bertino, Elisa and Won Kim. "Indexing Techniques for Queries on Nested Objects," *IEEE Transactions on Knowledge and Data Engineering*, *1*(2):196–214 (June 1989).
4. Cardenas, Alfonso F. and Dennis McLeod, editors. *Research Foundations in Object-Oriented and Semantic Database Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1990.
5. Carey, Michael J. and others. "Storage Management for Objects in EXODUS." Computer Sciences Department, University of Wisconsin, 1989.
6. Carey, Michael J. and others. "Using the EXODUS Storage Manager V1.2." Computer Sciences Department, University of Wisconsin, 1989.
7. Carey, Michael J. and others. "The EXODUS Extensible DBMS Project: An Overview." In Zdonik, Stanley B. and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–499, San Mateo, CA: Morgan Kaufmann, 1990.
8. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, *13*(6):377–387 (June 1970).
9. Colby, Latha S. "A Recursive Algebra and Query Optimization for Nested Relations." In Clifford, James, et al., editors, *Proceedings of the 1989 ACM-SIGMOD International Conference on the Management of Data*, pages 273–283, May 1989.
10. Dadam, P., et al. "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies." In Zaniolo, Carlo, editor, *Proceedings of the 1986ACM-SIGMOD International Conference on the Management of Data*, pages 356–367, May 1986.
11. Deshpande, Anand and Dirk Van Gucht. "An Implementation for Nested Relational Databases." In Bancilhon, Francois and David J. DeWitt, editors, *Proceedings of the Fourteenth International Conference on ' .ry Large Data Bases*, pages 76–87, August 1988.
12. Graefe, Goetz. "User Manual for the EXODUS Query Optimizer Generator." Oregon Graduate Center, Department of Computer Science and Engineering, February 1989.
13. Graefe, Goetz and David J. DeWitt. "The EXODUS Optimizer Generator." Computer Sciences Department, University of Wisconsin, 1989.
14. Hanson, Eric N. "An Initial Report on the Design of Ariel: A DBMS with an Integrated Production Rule System," *SIGMOD Record*, *18*(3) (September 1989).

15. Hartrum, Thomas C. "AFIT Department of Electrical and Computer Engineering System Development Documentation Guidelines and Standards." Draft 4, January 1989.

16. Heinz-Bernhard, Paul, et al. "Architecture and Implementation of the Darmstadt Database Kernel System." In Dayal, Umeshwar and Irv Traiger, editors, *Proceedings of the 1987ACM-SIGMOD International Conference on the Management of Data*, pages 196–207, May 1987.

17. Jaeschke, Gerhard and Hans-Jörg Schek. "Remarks on the Algebra of Non First Normal Form Relations." In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database systems*, pages 124–138, March 1982.

18. Korth, Henry F. and Abraham Silbershatz. *Database Systems Concepts*. New York: McGraw-Hill Book Company, 1986.

19. Kroenke, David. *Database Processing: Fundamentals, Design, Implementation.* Chicago: Science Research Associates, 1983.

20. Makinouchi, A. "A Consideration of Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model," *Proc. 3rd VLDB*, pages 447–453 (1977).

21. Mankus, Capt Michael A. *Design and Implementation of the Nested Relational Data Model Under the EXODUS Extensible Database System.* MS thesis, AFIT/GCS/ENG/89D-11, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

22. Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH 45433. *Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual (IDEF$_0$)*, June 1981.

23. Morris, Capt Gerald R. *A Comparison of a Relational and Nested-Relational IDEF$_0$ Data Model.* MS thesis, AFIT/GCE/ENG/89D-5, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1990.

24. Richardson, Joel E. and Michael J. Carey. "Persistence in the E Language: Issues and Implementation." Computer Sciences Department, University of Wisconsin, 1989.

25. Richardson, Joel E., et al. "The Design of the E Programming Language." Computer Sciences Department, University of Wisconsin, 1989.

26. Roth, Mark A., et al. "SQL/NF: A Query Language for ¬1NF Relational Databases," *Information Systems*, *12*(1):99–114 (1987).

27. Schek, H.-J. and Marc H. Scholl. "The Two Roles of Nested Relations in the DASDBS Project." In Abiteboul, S., et al., editors, *Nested Relations and Complex Objects in Databases (Lecture Notes in Computer Science 361)*, pages 50–68, Springer-Verlag, 1989.

28. Schnepf, Capt Craig W. *SQL/NF Translator for the Triton Nested Relational Database System.* MS thesis, AFIT/GCE/ENG/90D-05, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

29. Scholl, M., et al. "VERSO: A Database Machine Based On Nested Relations." In Abiteboul, S., et al., editors, *Nested Relations and Complex Objects in Databases (Lecture Notes in Computer Science 361)*, pages 27–49, Springer-Verlag, 1989.

30. Stonebraker, Michael, editor. *Readings in Database Systems*. San Mateo, CA: Morgan Kaufmann, 1988.

31. Stroustrup, Bjarne. *The C++ Programming Language*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1986.

32. Thomas, Stan J. and Patrick C. Fischer. "Nested Relational Structures." In Kanellakis, P. C., editor, *Advances in Computing Research, Volume 3: The Theory of Databases*, pages 269–307, JAI Press, 1985.

33. Zdonik, Stanley B. and David Maier, editors. *Readings in Object-Oriented Database Systems*. San Mateo, CA: Morgan Kaufmann, 1990.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE ACCESS AND OPERATOR METHODS FOR THE TRITON NESTED RELATIONAL DATABASE SYSTEM		5. FUNDING NUMBERS
6. AUTHOR(S) Tina M. Harvey, Capt, USAF		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583		8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/90D-06
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited		12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Unique database requirements in the realm of non-standard applications (such as computer-aided design (CAD), computer-aided software engineering (CASE), and office information systems (OIS)) have driven the development of new data models and database systems based on these new models. In particular, the goal of these new database systems is to exploit the advantages of complex data models that are more efficient (in terms of time and space) than their relational counterparts.
This research effort describes the design and implementation of the Triton nested relational database system, a prototype system based on the nested relational data model. Triton is intended to be used as the backed storage component for some non-standard application. To quickly prototype the system, the EXODUS extensible database system is used in the development of Triton.
The research presented in this document focuses on Triton's operator and access methods, and compares the performance of the nested relational model versus the relational model using these methods. In addition, the effectiveness of the EXODUS extensible database toolkit is evaluated.

14. SUBJECT TERMS data bases, nested relational databases, computer storage systems, database access methods, structured analysis			15. NUMBER OF PAGES 125
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL